Winter 2015

Lecture 8: Decrease & Conquer (continued)

Andrew P. Black

Department of Computer Science
Portland State University
Example: DFS traversal of undirected graph

DFS traversal stack:  

DFS tree:
Decrease by a Constant Factor

- binary search and bisection method (§12.4)
- exponentiation by squaring
- multiplication à la russe
Variable-size decrease

- Euclid’s algorithm
- Median (or percentile) by partition
- Nim-like games
Finding the Median

✧ The Median of an array of numbers is the “middle” number, when sorted.
✧ We can obviously find the median by sorting the array, and then picking the \( \left\lfloor \frac{n}{2} \right\rfloor \)th element.
✧ How much work is that (in average case)?

A. \( O(n) \)
B. \( O(n \ lg \ n) \)
C. \( O(n^2) \)
D. something else
Median in Linear Time?

✧ Can we do better?
  ▪ After all, sorting the whole array is more work than is needed to find the median

✧ Key insight: generalize the problem!
  ▪ Rather than seeking an algorithm for the $\left\lfloor \frac{n}{2} \right\rfloor$th element, let's look for the $k$th element, $k \in [1..n]$

Suppose that we have a way of partitioning the array at element with value $p$:

\[
\begin{array}{ccc}
\text{l} & \text{p} & \text{r} \\
\leq p & p & \geq p
\end{array}
\]

How can this help?
Suppose that we are looking for the 10th element, and:

\[ |A_{lo}| = 5 \]

\[ |A_p| = 1 \]

Then we can seek the 4th element of \( A_{hi} \) instead.

We have reduced the problem size by a variable amount, in this case \( |A_{lo}| + |A_p| = 6 \).
Suppose that we are looking for the 10th element, and:

- $|A_{lo}| = 28$
- Then we can seek the 10th element of $A_{lo}$ instead
- We have reduced the problem size by a variable amount, in this case $|A_p| + |A_{hi}|$
Suppose that we are looking for the 8th element, and:

- $|A_{lo}| = 6$
- $|A_p| = 2$

Then we can seek the 2nd element of $A_p$ instead.

We have now solved the problem, because all the elements of $A_p$ are $p$.
Variable-size decrease?

✧ What’s the connection?

- suppose that we have $A[1:20]$ and are looking for the 7\textsuperscript{th}-smallest element:
- run partition, find $s = 9$, say
- Where do we look for the 7\textsuperscript{th}-smallest element?
  
  A: $A[1..20]
  
  B: $A[1..8]
  
  C: $A[1..9]
  
  D: $A[10..20]
Variable-size decrease?

What’s the connection?

- suppose that we have $A[1:20]$ and are looking for the 7$^{th}$-smallest element:
- run partition, find $s = 3$, say
- Where do we look for the 7$^{th}$-smallest element?

A: $A[1..3]$
B: $A[1..4]$
C: $A[3..20]$
D: $A[4..20]$
What’s the Efficiency?

✧ Dasgupta’s analysis shows that:
    if we can do the partition in $O(n)$ time,
    then we can select the $k^{th}$ element in $O(n)$ time

✧ How can we do partition in $O(n)$ time?
  ➡ Lomuto Partition
  ➡ Hoare Partition
Lomuto Partition

✧ While algorithm is running:

<table>
<thead>
<tr>
<th>l</th>
<th>s</th>
<th>i</th>
<th>r</th>
</tr>
</thead>
<tbody>
<tr>
<td>p</td>
<td>&lt; p</td>
<td>≥ p</td>
<td>?</td>
</tr>
</tbody>
</table>

✧ Invariant:

- \( A[l] = p \land A[l+1..s] < p \land A[s+1..i-1] \geq p \land l \leq s < i \leq r \)

✧ Establish invariant initially:

- \( p \leftarrow A[l]; \ s \leftarrow l; \ i \leftarrow s+1 \)
  // makes \(< p\) interval and \(\geq p\) intervals both empty
I don't like Lumuto Partition

$p < p \geq p ?$
I don't like Lumuto Partition

✧ It does more swaps than necessary

\[
\begin{array}{c|c|c|c}
 l & s & i & r \\
p & <p & \geq p & ? \\
\end{array}
\]

Thursday, 22 October 2015
I don't like Lumuto Partition

✦ It does more swaps than necessary

✦ “half of the swap” is wasted

✦ It confuses students!
  ✦ Quicksort does not use the Lumuto Partition
Hoare Partition

- Classic algorithm of computing
- Not only linear, but peculiarly efficient!
- Tony Hoare won the Turing Award for Quicksort, which is based on this algorithm
  ... and some other things!
ALGORITHM 63
PARTITION
C. A. R. HOARE
Elliott Brothers Ltd., Borehamwood, Hertfordshire, Eng.

procedure partition (A,M,N,I,J); value M,N;
    array A; integer M,N,I,J;

comment I and J are output variables, and A is the array (with
subscript bounds M:N) which is operated upon by this procedure.
Partition takes the value X of a random element of the array A,
and rearranges the values of the elements of the array in such a
way that there exist integers I and J with the following properties:
M ≤ J < I ≤ N provided M < N
A[R] ≤ X for M ≤ R ≤ J
A[R] = X for J < R < I
A[R] ≥ X for I ≤ R ≤ N

The procedure uses an integer procedure random (M,N) which
chooses equiprobably a random integer F between M and N, and
also a procedure exchange, which exchanges the values of its two
parameters;

begin real X; integer F;
    F := random (M,N); X := A[F];
    I := M; J := N;
end

up: for I := I step 1 until N do
    if X < A[I] then go to down;
    I := N;
end

down: for J := J step -1 until M do
    if A[J]<X then go to change;
    J := M;
end

change: if I < J then begin exchange (A[I], A[J]);
        I := I + 1; J := J - 1;
        go to up
    end

else if I < F then begin exchange (A[I], A[F]);
        I := I + 1
    end

else if F < J then begin exchange (A[F], A[J]);
        J := J - 1
    end

end partition
ALGORITHM 63
PARTITION
C. A. R. Hoare
Elliott Brothers Ltd., Borehamwood, Hertfordshire, Eng.

procedure partition (A,M,N,I,J); value M,N;
array A; integer M,N,I,J;

comment I and J are output variables, and A is the array (with
subscript bounds M:N) which is operated upon by this procedure.
Partition takes the value X of a random element of the array A,
and rearranges the values of the elements of the array in such a
way that there exist integers I and J with the following properties:
M ≤ J < I ≤ N provided M < N
A[R] ≤ X for M ≤ R ≤ J
A[R] = X for J < R < I
A[R] ≥ X for I ≤ R ≤ N
The procedure uses an integer procedure random (M,N) which
chooses equiprobably a random integer F between M and N, and
also a procedure exchange, which exchanges the values of its two
parameters;

begin real X; integer F;
F := random (M,N); X := A[F];
I := M; J := N;
up: for I := I step 1 until N do
    if X < A [I] then go to down;
    I := N;
    down: for J := J step -1 until M do
        if A[J]<X then go to change;
        J := M;
change: if I < J then begin exchange (A[I], A[J]);
                I := I + 1; J := J - 1;
                go to up
        end
else if I < F then begin exchange (A[I], A[F]);
            I := I + 1
        end
else if F < J then begin exchange (A[F], A[J]);
                J := J - 1
        end;
end partition
ALGORITHM 63
PARTITION
C. A. R. Hoare
Elliott Brothers Ltd., Borehamwood, Hertfordshire, Eng.

procedure partition (A,M,N,I,J); value M,N;
array A; integer M,N,I,J;

comment I and J are output variables, and A is the array (with subscript bounds M:N) which is operated upon by this procedure. Partition takes the value $X$ of a random element of the array A, and rearranges the values of the elements of the array in such a way that there exist integers I and J with the following properties:

- $M \leq I < J \leq N$ provided $M < N$
- $A[R] \leq X$ for $M \leq R \leq J$
- $A[R] = X$ for $J < R < I$
- $A[R] \geq X$ for $I \leq R \leq N$

The procedure uses an integer procedure random $(M,N)$ which chooses equiprobably a random integer $F$ between $M$ and $N$, and also a procedure exchange, which exchanges the values of its two parameters;

begin real $X$; integer $F$;
$F :=$ random $(M,N)$; $X := A[F]$;
$I := M$; $J := N$;

up: for $I := I$ step 1 until $N$ do
if $X < A[I]$ then go to down;
$I := N$;

down: for $J := J$ step $-1$ until $M$ do
if $A[J] < X$ then go to change;
$J := M$;

change: if $I < J$ then begin exchange $(A[I], A[J])$;
$I := I + 1$; $J := J - 1$; go to up;
end
else if $I < F$ then begin exchange $(A[I], A[F])$;
$I := I + 1$
end
else if $F < J$ then begin exchange $(A[F], A[J])$;
$J := J - 1$
end;
end partition
Partition: CACM (Vol 4) July 1961

ALGORITHM 63
PARTITION
C. A. R. Hoare
Elliott Brothers Ltd., Borehamwood, Hertfordshire, Eng.

procedure partition (A,M,N,I,J); value M,N;
array A; integer M,N,I,J;

comment I and J are output variables, and A is the array (with subscript bounds M-N) which is operated upon by this procedure. Partition takes the value X of a random element of the array A, and rearranges the values of the elements of the array in such a way that there exist integers I and J with the following properties:
M ≤ J < I ≤ N provided M < N
A[R] ≤ X for M ≤ R ≤ J
A[R] = X for J < R < I
A[R] ≥ X for I ≤ R ≤ N

The procedure uses an integer procedure random (M,N) which chooses equiprobably a random integer F between M and N, and also a procedure exchange, which exchanges the values of its two parameters;

begin real X; integer F;
F := random (M,N); X := A[F];
I := M; J := N;
up: for I := I step 1 until N do
if X < A[I] then go to down;
I := N;
down: for J := J step -1 until M do
if A[J] < X then go to change;
J := M;
change: if I < J then begin exchange (A[I], A[J]);
I := I + 1; J := J - 1; go to up
end
else if I < F then begin exchange (A[I], A[F]);
I := I + 1
end
else if F < J then begin exchange (A[F], A[J]);
J := J - 1
end;
end partition
ALGORITHM 63
PARITION
C. A. R. Hoare
Elliott Brothers Ltd., Borehamwood, Hertfordshire, Eng.

procedure partition (A,M,N,I,J); value M,N;
array A; integer M,N,I,J;

comment I and J are output variables, and A is the array (with subscript bounds M-N) which is operated upon by this procedure. Partition takes the value X of a random element of the array A, and rearranges the values of the elements of the array in such a way that there exist integers I and J with the following properties:
M ≤ J < I ≤ N provided M < N
A[R] ≤ X for M ≤ R ≤ J
A[R] = X for J < R < I
A[R] ≥ X for I ≤ R ≤ N

The procedure uses an integer procedure random (M,N) which chooses equi-probably a random integer F between M and N, and also a procedure exchange, which exchanges the values of its two parameters;

begin real X; integer F;
F := random (M,N); X := A[F];
I := M, J := N;
up: for I := I step 1 until N do
if X < A[I] then go to down;
I := N;
down: for J := J step -1 until M do
if A[J] < X then go to change;
J := M;
change: if I < J then begin exchange (A[I], A[J]);
I := I + 1; J := J - 1;
go to up
end
else if I < F then begin exchange (A[I], A[F]);
I := I + 1
end
else if F < J then
begin exchange (A[F], A[J]);
J := J - 1
end;
end partition

Important features:
- random pivot
- double-ended search
- works in place
- two outputs
method partition(A, lo, hi) {
    def pivotIndex = randomBetween(lo) and (hi)
    def pivot = A[pivotIndex]
    var i := lo - 1
    var j := hi + 1
    while {
        do { i := i + 1 }
        while { (i <= hi).andAlso {A[i] <= pivot} }
        do { j := j - 1 }
        while { (j >= lo).andAlso {A[j] >= pivot} }
        i < j
    } do { exchange(A, i, j) }
    if (i < pivotIndex) then { exchange(A, i, pivotIndex) ; i := i + 1 }
    elseif (j > pivotIndex) then { exchange(A, pivotIndex, j) ; j := j - 1 }
    list.with(i, j)
}
Before partition begins:

```
  i
  lo
    p
  hi
  j
```
Before partition begins:

Leave elements that are already in the right place:
Before partition begins:

\[
\begin{array}{c}
\text{lo} \quad \text{hi} \\
\text{i} \quad \text{p} \\
\text{j}
\end{array}
\]

Leave elements that are already in the right place:

\[
\begin{array}{c}
\text{lo} \quad \text{hi} \\
\text{i} \quad \text{j} \\
\text{lo} \quad \text{hi}
\end{array}
\]

Now \( a[i] \geq p \geq a[j] \), so swap \( a[i] \) and \( a[j] \):

\[
\begin{array}{c}
\text{lo} \quad \text{hi} \\
\text{i} \quad \text{j} \\
\text{lo} \quad \text{hi}
\end{array}
\]
Before partition begins:

```
   i
lo p hi
```

Leave elements that are already in the right place:

```
   i
lo
   j
```

Now \( a[i] \geq p \geq a[j] \), so swap \( a[i] \) and \( a[j] \):

```
   i
lo \leq p \rightarrow
   j
```

Thursday, 22 October 2015
Before partition begins:

```
  i       p       j
  lo  |       |  hi
```

Leave elements that are already in the right place:

```
  i       j
  lo  |  hi
```

Now $a[i] \geq p \geq a[j]$, so swap $a[i]$ and $a[j]$:

```
  i       j
  lo  |  hi
```

$\leq p \rightarrow$  $\geq p \leftarrow$
Before partition begins:

Leave elements that are already in the right place:

Now $a[i] \geq p \geq a[j]$, so swap $a[i]$ and $a[j]$:

And continue ...
when do we stop?

And continue ...  

until i and j cross!

Thursday, 22 October 2015
when do we stop?

And continue ...

until $i$ and $j$ cross!

is this possible?
when do we stop?

And continue ...

until i and j cross!

is this possible?
when do we stop?

And continue ...

until i and j cross!

is this possible?
when do we stop?

And continue ...

until $i$ and $j$ cross!

is this possible?
when do we stop?

And continue ...

until i and j cross!

is this possible?
12 Coins

♙ This problem is originally stated as:
   ▪ You have a balance scale and 12 coins, 1 of which is counterfeit. The counterfeit weigh less or more than the other coins. Can you determine the counterfeit in 3 weightings, and tell if it is heavier or lighter?

♙ A harder and more general problem is:
   ▪ For some given $n > 1$, there are $(3^n - 3)/2$ coins, 1 of which is counterfeit. The counterfeit weigh less or more than the other coins. Can you state a priori $n$ weighting experiments with a balance, with which you determine the counterfeit coin, and tell if it is heavier or lighter?
Problem: Binary Insertion Sort

Binary insertion sort uses binary search to find an appropriate position to insert $A[i]$ among the previously sorted $A[0] \leq \ldots \leq A[i-1]$. In your work-groups, determine the worst-case efficiency class of this algorithm.

Hint: The order of growth of the worst-case number of key comparisons made by binary insertion sort can be obtained from formulas in Section 4.1 and Appendix A. For this algorithm, however, a key comparison is not the operation that determines the algorithm’s efficiency class. Which operation does?
Problem: Gray Code

Use the decrease-by-one technique (Algorithm BRGC) to generate the binary reflected Gray code for $n = 4$. 
Problem: Gray Code

Use the decrease-by-one technique (Algorithm BRGC) to generate the binary reflected Gray code for \( n = 4 \).

The list of bit strings in the binary reflexive Gray code for \( n = 3 \) given in the section is obtained by traversing the vertices of the three-dimensional cube by starting at 000 and following the arrows shown:

000 001 011 010 110 111 101 100.

We can obtain the binary reflexive Gray code for \( n = 4 \) as follows. Make two copies of the list of bit strings for \( n = 3 \); add 0 in front of each bit string in the first copy and 1 in front of each bit string in the second copy and then append the second list to the first in reversed order to obtain:

0 000 0001 0011 0010 0110 0111 0101 0100
1 100 1101 1111 1110 1010 1011 1001 1000

(Note that the last bit string differs from the first one by a single bit, too.)
Problem: Gray Code Algorithm

*Trace the following algorithm for generating the Binary Gray Code of order 4.*

Start with code = 0000

output code

for i = 1 to 15 do:

  b ← position of least significant 1 in binary rep of i

  code ← code XOR (bit b)

output code
Nim

✧ 1 pile of \( n \) chips
✧ Players take turns removing \( 1 \leq k \leq m \) chips
✧ The player removing the last chip wins

\( m = 4 \)
Nim

✩ 1 pile of $n$ chips
✩ Players take turns removing $1 \leq k \leq m$ chips
✩ The player removing the last chip wins

$$m = 4$$
Nim

✧ 1 pile of $n$ chips
✧ Players take turns removing $1 \leq k \leq m$ chips
✧ The player removing the last chip wins

$m = 4$
Nim

- 1 pile of \( n \) chips
- Players take turns removing \( 1 \leq k \leq m \) chips
- The player removing the last chip wins

\[ m = 4 \]
Nim

- 1 pile of $n$ chips
- Players take turns removing $1 \leq k \leq m$ chips
- The player removing the last chip wins

$m = 4$
Nim

* 1 pile of \( n \) chips
* Players take turns removing \( 1 \leq k \leq m \) chips
* The player removing the last chip wins

\[ m = 4 \]
Multiplication à la russe

\[ n \cdot m = \begin{cases} \frac{n}{2} \cdot 2m & \text{if } n \text{ is even} \\ \frac{n-1}{2} \cdot 2m + m & \text{if } n \text{ is odd} \end{cases} \]

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>50</td>
<td>65</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>130</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>260</td>
<td>(+130)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>520</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1,040</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>2,080</td>
<td>(+1040)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2,080</td>
<td></td>
<td>(+130 + 1040) = 3,250</td>
<td></td>
<td></td>
</tr>
<tr>
<td>50</td>
<td>65</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>130</td>
<td>130</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>260</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>520</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1,040</td>
<td>1,040</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>2,080</td>
<td>2,080</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3,250</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
You try it!

✧ multiply $37 \times 67$

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$37$</td>
<td>$67$</td>
</tr>
</tbody>
</table>
You try it!

✧ multiply $37 \times 67$

<table>
<thead>
<tr>
<th>$n$</th>
<th>$m$</th>
</tr>
</thead>
<tbody>
<tr>
<td>37</td>
<td>67</td>
</tr>
<tr>
<td>18</td>
<td></td>
</tr>
</tbody>
</table>
You try it!

✧ multiply $37 \times 67$

<table>
<thead>
<tr>
<th>$n$</th>
<th>$m$</th>
</tr>
</thead>
<tbody>
<tr>
<td>37</td>
<td>67</td>
</tr>
<tr>
<td>18</td>
<td>134</td>
</tr>
</tbody>
</table>
You try it!

✧ multiply $37 \times 67$

\[
\begin{array}{c|c|c}
 n & m \\
37 & 67 \\
18 & 134 & + 67 \\
\end{array}
\]