Final Tue 12-3

asgs: no extension.

today ten call stack

stack trace of C program
```c
#include <stdio.h>

int a(int x) {  
  return 2*x;
}

int b(int y) {  
  int u, v;
  u = a(y);
  v = u + y;
  return 2*v;
}

int main(void) {  
  printf("%d\n", a(3));  
  printf("%d\n", b(2));  
  return 0;
}
```
Recursion tree:

Main: 05

a: line 12
x = 3

Printf: line 12
"6 \n"

b: line 13
y = 2
u = 4
v = 5

Printf: line 13
"12 \n"

a: line 7
x = 2

trace of call stack:

(empty) main a main b main main b

a b main main b main (empty)
Snapshot of stack at Point *

<table>
<thead>
<tr>
<th>a</th>
<th>line 7, x = 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>line 13, y = 2, u = □, v = □</td>
</tr>
<tr>
<td>Main</td>
<td>OS</td>
</tr>
</tbody>
</table>

Quick sort:

See notes at:

http://www.soe.ucsc.edu/classes/cmasc212b
/Winter09/notes.html

Pages: 111-113

Also see Program Sort2 in
... /Winter09/Lecture
Void QuickSort (int * A, int p, int r) {
    int q;
    if (p < r) {
        q = Partition (A, p, r);
        QuickSort (A, p, q-1);
        QuickSort (A, q+1, r);
    }
}

Partition Re-arranges the Subarray A[p..r] and Returns an Index q such that:


The Element A[q] is called the Avor.

int Partition (A, p, r) {
    int l, j, x;
    x = A[r];
    l = p-1;
    for (j = p; j < r; j++) {
        if (A[j] > x) {
            l++;
            swap (A, l, j);
        }
    }
    swap (A, l+1, r);
    return (l+1);
}
WE USE A HELPER FUNCTION SWAP WHICH EXCHANGES ARRAY ELEMENTS.

```c
void swap (int *a, int i, int j) {
    int temp;
    temp = a[i];
    a[i] = a[j];
    a[j] = temp;
}
```

**Partition:** Picks an element in \( A[p...r] \) to be the so-called pivot. In our case, the pivot is \( x = a[i] \). (In the book it \( a[p] \).) Partition then divides the subarray \( A[p...r] \) into two sections: those elements less than or equal to \( x \) and those which are greater than \( x \). It then swaps the pivot into a position between the two sections, then returns the index of the pivot.

```
\[
\begin{array}{l}
p & q \\
\hline
x & r \\
\end{array}
\]
```

**Thus:** \( A[p...q-1] \leq a[q] < A[q+1...r] \) won return of partition. All that is left is to sort subarrays \( A[p...q-1] \) and \( A[q+1...r] \) recursively, which Quicksort does.
Examining the code for partition we see that the FOR loop maintains the invariant:

\[ p \quad i \quad i+1 \quad \ldots \quad j-1 \quad j \quad \ldots \quad n \quad x \]

\[ \leq x \quad > x \quad \text{UNKNOWN} \]

i.e. \( A[p \ldots i] \leq x < A[i+1 \ldots j-1] \). This invariant is certainly true before the first iteration, since both sub-sequences are empty. It's not hard to see that the invariant is maintained from one iteration to another. When \( j = p \) we have:

\[ i \quad i+1 \quad \ldots \quad j-1 \quad x \]

\[ \leq x \quad > x \]

The single swap \( A[i+1] \leftrightarrow A[i] \) gives:

\[ \ldots \quad i \quad i+1 \quad x \quad \ldots \quad r \]

\[ \leq x \quad > x \]

Then we return the index \( i+1 \).

Quick sort is dual in some sense to merge sort. Both are divide-and-conquer algorithms. Merge sort does its real work in the recombine step, while quick sort does it in the divide step.