CS 2420: Algorithm Analysis

Dr. Tsung-Wei (TW) Huang

Department of Electrical and Computer Engineering

University of Utah, Salt Lake City, UT

Outline

- In this topic, we will examine code to determine the run time of various operations

- We will introduce machine instructions

- We will calculate the run times of:
  - Operators **+**, **-**, **=**, **+=**, **++**, etc.
  - Control statements **if**, **for**, **while**, **do-while**, **switch**
  - Functions
  - Recursive functions

Motivation

- The goal of algorithm analysis is to take a block of code and determine the asymptotic run time or asymptotic memory requirements based on various parameters
  - Given an array of size n:
    - Selection sort requires $\Theta(n^2)$ time
    - Merge sort, quick sort, and heap sort all require $\Theta(n \ln(n))$ time
  - However:
    - Merge sort requires $\Theta(n)$ additional memory
    - Quick sort requires $\Theta(\ln(n))$ additional memory
    - Heap sort requires $\Theta(1)$ memory

Motivation

- The asymptotic complexity of algorithms indicates its ability to scale
  - Suppose we are sorting an array of size n

- Selection sort has a run time of $\Theta(n^2)$
  - $2n$ entries requires $(2n)^2 = 4n^2$
    - Four times as long to sort
  - $10n$ entries requires $(10n)^2 = 100n^2$
    - One hundred times as long to sort

Motivation

- The other sorting algorithms have $\Theta(n \ln(n))$ run times
  - $2n$ entries require $(2n) \ln(2n) = (2n) (\ln(n) + 1) = 2(n \ln(n)) + 2n$
  - $10n$ entries require $(10n) \ln(10n) = (10n) (\ln(n) + 1) = 10(n \ln(n)) + 10n$

- In each case, it requires $\Theta(n)$ more time

- However:
  - Merge sort will require twice and 10 times as much memory
  - Quick sort will require one or four additional memory locations
  - Heap sort will not require any additional memory

Motivation

- If we are storing objects which are not related, the hash table has, in many cases, optimal characteristics:
  - Many operations are $\Theta(1)$
  - *I.e.*, the run times are independent of the number of objects being stored

- If we are required to store both objects and relations, both memory and time will increase
  - Our goal will be to minimize this increase

Motivation

- To properly investigate the determination of run times asymptotically:
  - We will begin with machine instructions
  - Discuss operations
  - Control statements
    - Conditional statements and loops
  - Functions
  - Recursive functions

Machine Instructions

- Given any processor, it is capable of performing only a limited number of operations

- These operations are called *instructions*

- The collection of instructions is called the *instruction set*
  - The exact set of instructions differs between processors
  - MIPS, ARM, x86, 6800, 68k

Machine Instructions

- Any instruction runs in a fixed amount of time (an integral number of CPU cycles)
- An example on the Coldfire is:

  `0x06870000000F`

  which adds 15 to the 7th data register
- As humans are not good at hex, this can be programmed in assembly language as

```
ADDI.L #$F, D7
```

Machine Instructions

- Assembly language has an almost one-to-one translation to machine instructions
  - Assembly language is a low-level programming language

- Other programming languages are higher-level:

  Fortran, Pascal, Matlab, Java, C++, and C#

- The adjective “high” refers to the level of abstraction:
  - Java, C++, and C# have abstractions such as OO
  - Matlab and Fortran have operations which do not map to relatively small number of machine instructions:

```
>> 1.27^2.9                          % 1.27**2.9 in Fortran
      2.0000036616123606774
```

Machine Instructions

- The C programming language (C++ without objects and other abstractions) can be referred to as a mid-level programming language
  - There is abstraction, but the language is closely tied to the standard capabilities
  - There is a closer relationship between operators and machine instructions

- Consider the operation `a += b;`
  - Assume that the compiler has already has the value of the variable **a** in register **D1** and perhaps **b** is a variable stored at the location stored in address register **A1**, this is then converted to the single instruction

```
ADD (A1), D1
```

Operators

- Because each machine instruction can be executed in a fixed number of cycles, we may assume each operation requires a fixed number of cycles
  - The time required for any operator is $\Theta(1)$ including:
    - Retrieving/storing variables from memory
    - Variable assignment `=`
    - Integer operations `+ - * / % ++ --`
    - Logical operations `&& || !`
    - Bitwise operations `& | ^ ~`
    - Relational operations `== != < <= => >`
    - Memory allocation and deallocation `new delete`

Operators

- Of these, memory allocation and deallocation are the slowest by a significant factor
  - A quick test on a linux machine shows a factor of over 100
  - They require communication with the operation system
  - This does not account for the time required to call the constructor and destructor

- Note that after memory is allocated, the constructor is run
  - The constructor may not run in $\Theta(1)$ time

Blocks of Operations

- Each operation runs in $\Theta(1)$ time and therefore any fixed number of operations also run in $\Theta(1)$ time, for example:

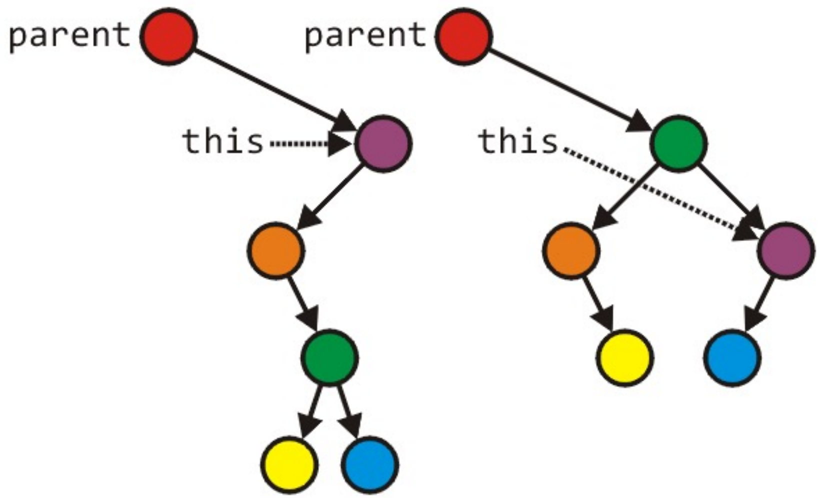
```
// Swap variables a and b
int tmp = a;
a = b;
b = tmp;

// Update a sequence of values
++index;
prev_modulus = modulus;
modulus = next_modulus;
next_modulus = modulus_table[index];
```

Blocks of Operations

- Seldom will you find large blocks of operations without any additional control statements

- This example rearranges an AVL tree structure

```
 Tree_node *lrl = left->right->left;
Tree_node *lrr = left->right->right;
parent = left->right;
parent->left  = left;
parent->right = this;
left->right = lrl;
left = lrr;
```



- Run time: $\Theta(1)$

Blocks in Sequence

- Suppose you have now analyzed a number of blocks of code run in sequence

```
template <typename T>
void update_capacity( int delta ) {
   T *array_old = array;
   int capacity_old = array_capacity;
   array_capacity += delta;
   array = new T[array_capacity];

   for ( int i = 0; i < capacity_old; ++i ) {
          array[i] = array_old[i];
   }

   delete[] array_old;
}
```

$\Theta(1)$

$\Theta(n)$

$\Theta(1)$ or $\Omega(n)$

- To calculate the total run time, add the entries: $\Theta(1 + n + 1) = \Theta(n)$

Control Statements

- Next we will look at the following control statements

- These are statements which potentially alter the execution of instructions
  - Conditional statements

    `if`, **`switch`**
  - Condition-controlled loops

    `for`, **`while`**, **`do-while`**
  - Count-controlled loops

    ```
    for i from 1 to 10 do ... end do;       # Maple
    ```
  - Collection-controlled loops

    ```
    foreach ( int i in array ) { ... }      // C#
    ```

Control Statements

- Given any collection of nested control statements, it is always necessary to work inside out
  - Determine the run times of the inner-most statements and work your way out

Control Statements

- Given

```
if ( condition ) {
    // true body
} else {
    // false body
}
```

- The run time of a conditional statement is:
  - the run time of the condition (the test), plus
  - the run time of the body which is run
- In most cases, the run time of the condition is $\Theta(1)$

Control Statements

- In some cases, it is easy to determine which statement must be run:

```
int factorial ( int n ) {
        if ( n == 0 ) {
                return 1;
        } else {
                return n * factorial ( n – 1 );
        }
}
```

Control Statements

- In others, it is less obvious
  - Find the maximum entry in an array:

```
int find_max( int *array, int n ) {
    max = array[0];

    for ( int i = 1; i < n; ++i ) {
        if ( array[i] > max ) {
            max = array[i];
        }
    }

    return max;
}
```

Analysis of Statements

- In this case, we don’t know

- If we had information about the distribution of the entries of the array, we may be able to determine it
  - if the list is sorted (ascending) it will always be run
  - if the list is sorted (descending) it will be run once
  - if the list is uniformly randomly distributed, then???

Condition-controlled Loops

- The C++ for loop is a condition controlled statement:

```
for ( int i = 0; i < N; ++i ) {
        // ...
}
```

is identical to

```
int i = 0;                    // initialization
while ( i < N ) {             // condition
    // ...
    ++i;                      // increment
}
```

Condition-controlled Loops

- The initialization, condition, and increment usually are single statements running in $\Theta(1)$

```
for ( int i = 0; i < N; ++i ) {
        // ...
}
```

Condition-controlled Loops

- The initialization, condition, and increment statements are usually $\Theta(1)$

  For example,

```
for ( int i = 0; i < n; ++i ) {
    // ...
}
```

- Assuming there are no break or return statements in the loop, the run time is $\Theta(n)$

Condition-controlled Loops

- If the body does not depend on the variable (in this example, `i`), then the run time of

```
for ( int i = 0; i < n; ++i ) {
    // code which is Theta(f(n))
}
```

  is $\Theta(n\ \mathrm{f}(n))$

- If the body is $\Theta(\mathrm{f}(n))$, then the run time of the loop is $\Theta(n\ \mathrm{f}(n))$

Condition-controlled Loops

- For example,

```
int sum = 0;
for ( int i = 0; i < n; ++i ) {
    sum += 1;      Theta(1)
}
```

- This code has run time

$$\Theta(n \cdot 1) = \Theta(n)$$

Condition-controlled Loops

- Another example example,

```
int sum = 0;
for ( int i = 0; i < n; ++i ) {
    for ( int j = 0; j < n; ++j ) {
        sum += 1;     Theta(1)
    }
}
```

- The previous example showed that the inner loop is $\Theta(n)$, thus the outer loop is

$$\Theta(n \cdot n) = \Theta(n^2)$$

Analysis of Repetition Statements

- Suppose with each loop, we use a linear search an array of size m:

```
for ( int i = 0; i < n; ++i ) {
    // search through an array of size m
    // Θ( m );
}
```

- The inner loop is $\Theta(m)$ and thus the outer loop is

$$\Theta(n\ m)$$

Conditional Statements

Consider this example

```
void Disjoint_sets::clear() {
    if ( sets == n ) {
        return;
    }

    max_height = 0;
    num_disjoint_sets = n;

    for ( int i = 0; i < n; ++i ) {
        parent[i] = i;
        tree_height[i] = 0;
    }
}
```

$\Theta(1)$

$\Theta(1)$

$\Theta(n)$

$\Theta(1)$

$$\mathrm{T}_{\text{clear}}(n) = \begin{cases} \Theta(1) & sets = n \\ \Theta(n) & \text{otherwise} \end{cases}$$

Analysis of Repetition Statements

- If the body does depends on the variable (in this example, `i`), then the run time of

```
for ( int i = 0; i < n; ++i ) {
    // code which is Theta(f(i,n))
}
```

is $$\Theta\left(1 + \sum_{i=0}^{n-1}\left(1 + f(i,n)\right)\right)$$

Analysis of Repetition Statements

- For example,

```
int sum = 0;
for ( int i = 0; i < n; ++i ) {
    for ( int j = 0; j < i; ++j ) {
        sum += i + j;
    }
}
```

- The inner loop is $\Theta(1 + i(1 + 1)) = \Theta(i)$ hence the outer is

$$\Theta\left(1+\sum_{i=0}^{n-1}(1+i)\right)=\Theta\left(1+n+\sum_{i=0}^{n-1}i\right)=\Theta\left(1+n+\frac{n(n-1)}{2}\right)=\Theta\left(n^2\right)$$

Analysis of Repetition Statements

- As another example:

```
int sum = 0;
for ( int i = 0; i < n; ++i ) {
    for ( int j = 0; j < i; ++j ) {
        for ( int k = 0; k < j; ++k ) {
            sum += i + j + k;
        }
    }
}
```

- From inside to out:

$\Theta(1)$
$\Theta(j)$
$\Theta(i^2)$
$\Theta(n^3)$

Control Statements

- Switch statements appear to be nested if statements:

```
switch( i ) {
      case 1:    /* do stuff */ break;
      case 2:    /* do other stuff */ break;
      case 3:    /* do even more stuff */ break;
      case 4:    /* well, do stuff */ break;
      case 5:    /* tired yet? */ break;
      default:   /* do default stuff */
}
```

Control Statements

- Thus, a switch statement would appear to run in $\Theta(n)$ time where n is the number of cases, the same as nested if statements
  - Why then not use:

```
if ( i == 1 ) { /* do stuff */ }
else if ( i == 2 ) { /* do other stuff */ }
else if ( i == 3 ) { /* do even more stuff */ }
else if ( i == 4 ) { /* well, do stuff */ }
else if ( i == 5 ) { /* tired yet? */ }
else { /* do default stuff */ }
```

Control Statements

- Question:

  Why would you introduce something into
  programming language which is redundant?

- There are reasons for this:
  - your name is Larry Wall and you are creating the Perl (**not** PERL) programming language
  - you are introducing software engineering constructs, for example, classes

Control Statements

- However, switch statements were included in the original C language... why?

- First, you may recall that the cases must be actual values, either:
  - integers
  - characters

- For example, you cannot have a case with a variable, e.g.,

```
case n: /* do something */ break;  //bad
```

Control Statements

- The compiler looks at the different cases and calculates an appropriate jump

- For example, assume:
  - the cases are 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
  - each case requires a maximum of 24 bytes (for example, six instructions)

- Then the compiler simply makes a jump size based on the variable, jumping ahead either 0, 24, 48, 72, ..., or 240 instructions

Serial Statements

Suppose we run one block of code followed by another block of code

Such code is said to be run *serially*

If the first block of code is $\Theta(\mathrm{f}(n))$ and the second is $\Theta(\mathrm{g}(n))$, then the run time of two blocks of code is

$$\Theta(\ \mathrm{f}(n) + \mathrm{g}(n)\ )$$

which usually (for algorithms not including function calls) simplifies to one or the other

Serial Statements

- Consider the following two problems:
  - search through a random list of size n to find the maximum entry, and
  - search through a random list of size n to find if it contains a particular entry

- What is the proper means of describing the run time of these two algorithms?

Serial Statements

- Searching for the maximum entry requires that each element in the array be examined, thus, it must run in $\Theta(n)$ time

- Searching for a particular entry may end earlier: for example, the first entry we are searching for may be the one we are looking for, thus, it runs in $\mathrm{O}(n)$ time
  - O notation represents the strict upper-bound of an algorithm
  - Θ notation represents the strict bound of an algorithm
  - Ω notation represents the strict lower-bound of an algorithm

Serial Statements

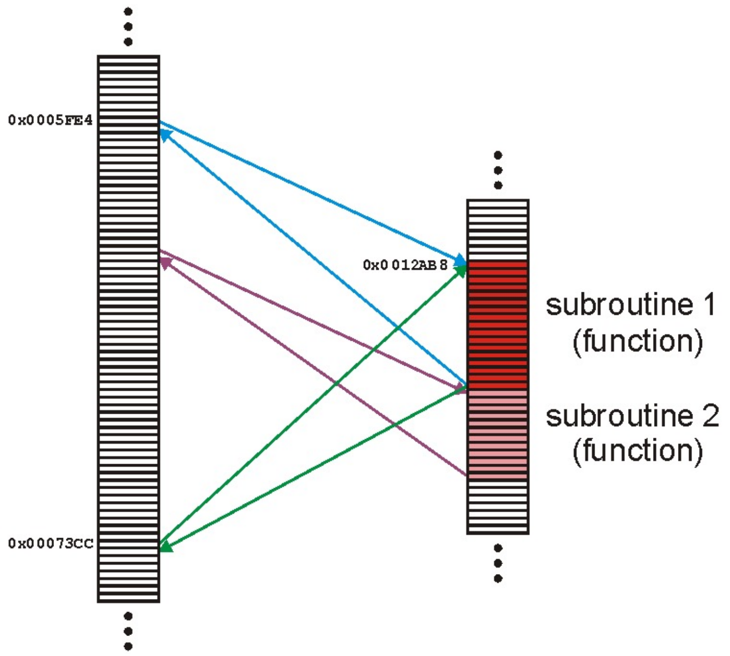
- Therefore:
  - if the leading term is big-$\mathbf{\Theta}$, then the result must be big-$\mathbf{\Theta}$, otherwise
  - if the leading term is big-$\mathbf{O}$, we can say the result is big-$\mathbf{O}$

- For example,

$$\mathbf{O}(n) + \mathbf{O}(n^2) + \mathbf{O}(n^4) = \mathbf{O}(n + n^2 + n^4) = \mathbf{O}(n^4)$$
$$\mathbf{O}(n) + \mathbf{\Theta}(n^2) = \mathbf{\Theta}(n^2)$$
$$\mathbf{O}(n^2) + \mathbf{\Theta}(n) = \mathbf{O}(n^2)$$
$$\mathbf{O}(n^2) + \mathbf{\Theta}(n^2) = \mathbf{\Theta}(n^2)$$

Functions

- A function (or subroutine) is code which has been separated out, either to:
  - and repeated operations
    - e.g., mathematical functions
  - group related tasks
    - e.g., initialization



Functions

- Because a subroutine (function) can be called from anywhere, we must:
  - prepare the appropriate environment
  - deal with arguments (parameters)
  - jump to the subroutine
  - execute the subroutine
  - deal with the return value
  - clean up

Functions

- Fortunately, this is such a common task that all modern processors have instructions that perform most of these steps in one instruction

- Thus, we will assume that the overhead required to make a function call and to return is $\Theta(1)$

Functions

- Because any function requires the overhead of a function call and return, we will always assume that

$$T_f = \Omega(1)$$

- That is, it is impossible for any function call to have a zero run time

Functions

- Thus, given a function $\mathrm{f}(n)$ (the run time of which depends on n) we will associate the run time of $\mathrm{f}(n)$ by some function $\mathrm{T}_{\mathrm{f}}(n)$
  - We may write this to $\mathrm{T}(n)$

- Because the run time of any function is at least $\mathbf{O}(1)$, we will include the time required to both call and return from the function in the run time

Functions

```
void Disjoint_sets::set_union( int m, int n ) {
        m = find( m );
        n = find( n );

        if ( m == n ) {
                return;
        }

        --num_disjoint_sets;

        if ( tree_height[m] >= tree_height[n] ) {
            parent[n] = m;

            if ( tree_height[m] == tree_height[n] ) {
                ++( tree_height[m] );
                max_height = std::max( max_height, tree_height[m] );
            }
        } else {
            parent[m] = n;
        }
}
```

$2T_{find}$

$T_{set\_union} = 2T_{find} + \Theta(1)$

$\Theta(1)$

Recursive Functions

- A function is relatively simple (and boring) if it simply performs operations and calls other functions

- Most interesting functions designed to solve problems usually end up calling themselves
  - Such a function is said to be *recursive*

Recursive Functions

- As an example, we could implement the factorial function recursively:

```
int factorial( int n ) {
    if ( n <= 1 ) {
        return 1;
    } else {
        return n * factorial( n - 1 );
    }
}
```

$$\Theta(1)$$

$$\mathrm{T}_{!}(n-1)+\Theta(1)$$

Recursive Functions

- Thus, we may analyze the run time of this function as follows:

$$\mathrm{T}_!(n) = \begin{cases} \mathbf{\Theta}(1) & n \le 1 \\ \mathrm{T}_!(n-1) + \mathbf{\Theta}(1) & n > 1 \end{cases}$$

- We don’t have to worry about the time of the conditional ($\mathbf{\Theta}(1)$) nor is there a probability involved with the conditional statement

Recursive Functions

- The analysis of the run time of this function yields a recurrence relation:

$$\mathrm{T}_!(n) = \mathrm{T}_!(n-1) + \Theta(1) \qquad \mathrm{T}_!(1) = \Theta(1)$$

Recursive Functions

If $k = n - 1$ then

$$\begin{aligned} T_{!}(n) &= T_{!}(n - (n - 1)) + n - 1 \\ &= T_{!}(1) + n - 1 \\ &= 1 + n - 1 = n \end{aligned}$$

Thus, $T_{!}(n) = \Theta(n)$

Recursive Functions

- Suppose we want to sort a array of n items

- We could:
  - go through the list and find the largest item
  - swap the last entry in the list with that largest item
  - then, go on and sort the rest of the array

- This is called *selection sort*

Recursive Functions

```
void sort( int * array, int n ) {
    if ( n <= 1 ) {
        return;                          // special case:  0 or 1 items are always sorted
    }

    int posn = 0;                        // assume the first entry is the smallest
    int max = array[posn];

    for ( int i = 1; i < n; ++i ) {     // search through the remaining entries
        if ( array[i] > max ) {          // if a larger one is found
            posn = i;                    // update both the position and value
            max = array[posn];
        }
    }

    int tmp = array[n - 1];              // swap the largest entry with the last
    array[n - 1] = array[posn];
    array[posn] = tmp;

    sort( array, n – 1 );                // sort everything else
}
```

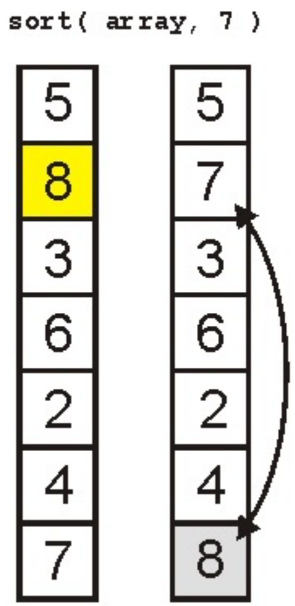
Recursive Functions

- We could call this function as follows:

```
    int array[7] = {5, 8, 3, 6, 2, 4, 7};
    sort( array, 7 );   // sort an array of seven items
```

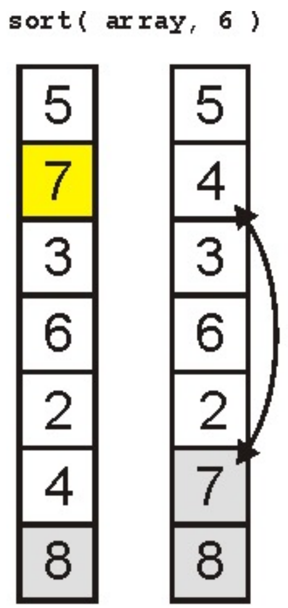
Recursive Functions

- The first call finds the largest element



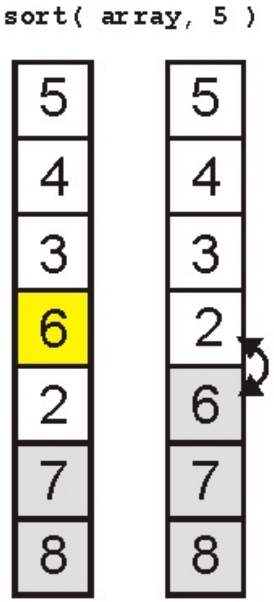
Recursive Functions

- The next call finds the 2^{nd}-largest element



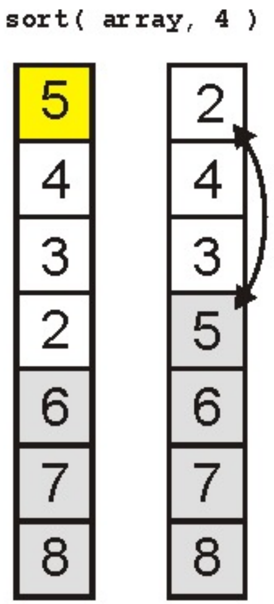
Recursive Functions

- The third finds the 3rd-largest



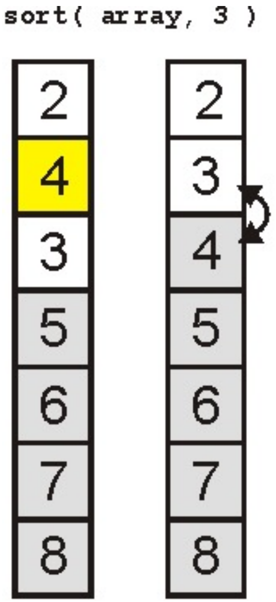
Recursive Functions

- And the 4^{th}



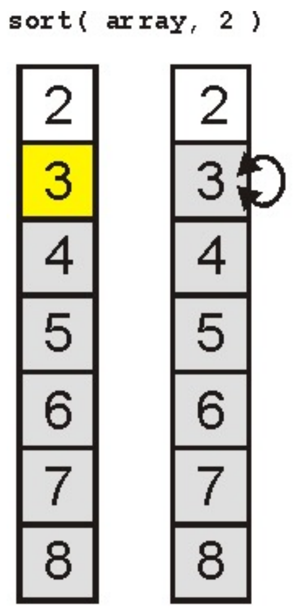
Recursive Functions

- And the 5th



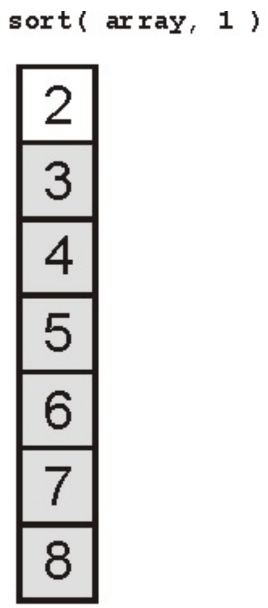
Recursive Functions

- Finally the 6th



Recursive Functions

- And the array is sorted:



Recursive Functions

- Analyzing the function, we get:

```
void sort( int * array, int n ) {
    if ( n <= 1 ) {
        return;
    }

    int posn = 0;
    int max = array[posn];

    for ( int i = 1; i < n; ++i ) {
        if ( array[i] > max ) {
            posn = i;
            max = array[posn];
        }
    }

    int tmp = array[n - 1];
    array[n - 1] = array[posn];
    array[posn] = tmp;

    sort( array, n - 1 );
}
```

$\mathrm{T}(0) = \mathrm{T}(1) = \Theta(1)$

$\Theta(1)$

$\Theta(1)$ $\quad \Theta(n)$

$\Theta(1)$

$\mathrm{T}(n-1)$

$$\begin{aligned}\mathrm{T}(n) &= \Theta(1) + \Theta(n) + \Theta(1) + \mathrm{T}(n-1)\\ &= \mathrm{T}(n-1) + \Theta(n)\end{aligned}$$

Recursive Functions

- Thus, replacing each symbol with a representative, we are required to solve the recurrence relation

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

The easy way to solve this is with Maple:

```
> rsolve( {T(n) = T(n - 1) + n, T(1) = 1}, T(n) );
```

$$-1 - n + (n+1)\left(\frac{n}{2} + 1\right)$$

```
> expand( % );
```

$$\frac{1}{2}n + \frac{1}{2}n^2$$

Recursive Functions

- Consequently, the sorting routine has the run time

$$\mathrm{T}(n) = \Theta(n^2)$$

- To see this by hand, consider the following

$$\begin{aligned}
\mathrm{T}(n) &= \mathrm{T}(n-1) + n \\
&= \big(\mathrm{T}(n-2) + (n-1)\big) + n \\
&= \mathrm{T}(n-2) + n + (n-1) \\
&= \mathrm{T}(n-3) + n + (n-1) + (n-2) \\
&\vdots \\
&= \mathrm{T}(1) + \sum_{i=2}^{n} i = 1 + \sum_{i=2}^{n} i = \sum_{i=1}^{n} i = \frac{n(n+1)}{2}
\end{aligned}$$

Recursive Functions

- Consider, instead, a binary search of a sorted list:
  - Check the middle entry
  - If we do not find it, check either the left- or right-hand side, as appropriate

Thus, $\mathrm{T}(n) = \mathrm{T}((n - 1)/2) + \Theta(1)$

Recursive Functions

- Also, if $n = 1$, then $\mathrm{T}(1) = \Theta(1)$

- Thus we have to solve:

$$\mathrm{T}(n) = \begin{cases} 1 & n = 1 \\ \mathrm{T}\left(\dfrac{n-1}{2}\right) + 1 & n > 1 \end{cases}$$

- Solving this can be difficult, in general, so we will consider only special values of n

- Assume $n = 2^k - 1$ where k is an integer

  Then $(n - 1)/2 = (2^k - 1 - 1)/2 = 2^{k-1} - 1$

Recursive Functions

- For example, searching a list of size 31 requires us to check the center

- If it is not found, we must check one of the two halves, each of which is size 15

$$31 = 2^5 - 1$$

$$15 = 2^4 - 1$$

Recursive Functions

- Thus, we can write

$$
\begin{aligned}
\mathrm{T}(n) &= \mathrm{T}(2^k - 1) \\
&= \mathrm{T}\left(\frac{2^k - 1 - 1}{2}\right) + 1 \\
&= \mathrm{T}(2^{k-1} - 1) + 1 \\
&= \mathrm{T}\left(\frac{2^{k-1} - 1 - 1}{2}\right) + 1 + 1 \\
&= \mathrm{T}(2^{k-2} - 1) + 2 \\
&\vdots
\end{aligned}
$$

Recursive Functions

- Notice the pattern with one more step:

$$
\begin{aligned}
&= \mathrm{T}(2^{k-1} - 1) + 1 \\
&= \mathrm{T}\left(\frac{2^{k-1} - 1 - 1}{2}\right) + 1 + 1 \\
&= \mathrm{T}(2^{k-2} - 1) + 2 \\
&= \mathrm{T}(2^{k-3} - 1) + 3 \\
&\vdots
\end{aligned}
$$

Recursive Functions

- Thus, in general, we may deduce that after $k - 1$ steps:

$$\begin{aligned} \mathrm{T}(n) &= \mathrm{T}(2^k - 1) \\ &= \mathrm{T}(2^{k-(k-1)} - 1) + k - 1 \\ &= \mathrm{T}(1) + k - 1 = k \end{aligned}$$

because $\mathrm{T}(1) = 1$

Recursive Functions

- Thus, $T(n) = k$, but $n = 2^k - 1$
- Therefore $k = \lg(n + 1)$
- However, recall that $f(n) = \Theta(g(n))$ if $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c$ for $0 < c < \infty$

$$\lim_{n\to\infty} \frac{\lg(n+1)}{\ln(n)} = \lim_{n\to\infty} \frac{\frac{1}{(n+1)\ln(2)}}{\frac{1}{n}} = \lim_{n\to\infty} \frac{n}{(n+1)\ln(2)} = \frac{1}{\ln(2)}$$

- Thus, $T(n) = \Theta(\lg(n + 1)) = \Theta(\ln(n))$

Summary

- In these slides we have looked at:
  - The run times of
    - Operators
    - Control statements
    - Functions
    - Recursive functions
  - We have also defined best-, worst-, and average-case scenarios

- We will be considering all of these each time we inspect any algorithm used in this class