# Recurrences

COMP 215

# Analysis of Iterative Algorithms

```
//return the location of the item matching x, or 0 if
//no such item is found.

index SequentialSearch(keytype[] S, int n, keytype x)
{
    index location = 1;
    while (location <= n && S[location] != x)
        location++
    if (location > n)
        location = 0;
    return location;
}
```

- It is straightforward to figure out the complexity here.
- Count the number of times the basic operation occurs, accounting for loops.

# Analysis of Recursive Algorithms

```
//factorial function

int fact(int n) {
   if (n == 0)
      return 1;
   else
      return n * fact(n-1);
}
```

- The situation here is trickier.
- How many times does the basic operation (multiplication) occur?

# Analysis of Recursive Algorithms

```
//factorial function

int fact(int n) {
   if (n == 0)
      return 1;
   else
      return n * fact(n-1);
}
```

- The situation here is trickier.
- How many times does the basic operation (multiplication) occur?
- Easily described in terms of a recurrence: $t_n = t_{n-1} + 1$
- Closed form?

# Inductive Proof...

- $t_0 = 0$
- $t_n = t_{n-1} + 1$

- Candidate solution: $t_n = n$

# Exercise

```
int newfun(int n) {
   a = 0;
   if (n == 0)
      return 1;
   else
      a +=  newfun(n-1);
      a +=  newfun(n-1);
      for (i = 1; i <=n; i++)
         a = a * n;
      return a;
}
```

- Assuming multiplication is the basic operation, develop a recurrence.

# Binary Search

```
//return the location of the item matching x, or 0 if //no such
item is found.  S must be sorted.

index BinarySearch(keytype[] S, int low, int high, keytype x)
{
    index mid;

    if (low > high)
        return 0;
    else {
        mid = floor( (low + high) / 2 )
        if (x == S[mid])
            return mid;
        else if (x < S[mid])
            return BinarySearch(low, mid – 1);
        else
            return BinarySearch(mid + 1, high);
    }
}
```

# Another Inductive Proof

- The recurrence is:

$$t_n = t_{\lfloor n/2 \rfloor} + 1$$
$$t_1 = 1$$

- For the same of simplicity, assume that n is a power of 2.

$$t_n = t_{n/2} + 1$$
$$t_1 = 1$$

- Candidate solution?

# Handling Non-Powers of Two (or b)

- For binary search we were able to exactly determine the complexity, as long as $n$ was a power of 2: $\lg n + 1$.
- It would be nice to be able to say *something* about binary search even if $n$ is not a power of 2.
- E.g. $T(n) \in \Theta(\lg n)$ for all $n$.

- First some definitions...

# Definitions

- A complexity function $f(n)$ is strictly increasing if $f(n)$ always gets larger as $n$ gets larger.
  - That is, if $n_1 > n_2$, then $f(n_1) > f(n_2)$.

- A complexity function $f(n)$ is non-decreasing if $f(n)$ never gets smaller as $n$ gets larger.
  - That is, if $n_1 > n_2$, then $f(n_1) >= f(n_2)$.

- Reminder: A complexity function can be any function that maps positive integers to non-negative reals.

# More Definitions

- A complexity function *f(n)* is eventually non-decreasing if for all n past some point the function never gets smaller as n gets larger.
  - That is, there exists an *N* such that if $n_1 > n_2 > N$ then $f(n_1) >= f(n_2)$.

- A complexity function *f(n)* is smooth if *f(n)* is eventually non-decreasing and if $f(2n) \in \Theta(f(n))$ .

- (Note that this is not the same as the calculus definition of smoothness)

# Finally, The Theorem

- let $b >= 2$ be an integer, let $f(n)$ be a smooth complexity function, and let $T(n)$ be an eventually non-decreasing complexity function. If

$$T(n) \in \Theta(f(n)) \qquad \textit{for } n \textit{ a power of } b\textit{, then}$$

$$T(n) \in \Theta(f(n))$$

- The same implication holds if $\Theta$ is replaced by "big O", $\Omega$, or "small o".

# Binary Search is in $\Theta(lg\, n)$

- We already know that W(n)=lg n +1, if n is a power of 2.
- We need to show that
  - lg *n* is smooth and that,
    - pretty easy.
  - $t_n = t_{\lfloor n/2 \rfloor} + 1$ is eventually non-decreasing.
    - requires induction.

# What if You Don't Have a Guess?

- Backward substitution...
- Recursion trees – particularly for divide and conquer algorithms.
- Recursion tree for $t_n = 2t_{n/2} + n$ ...
- “Cookbook” solutions.

# General Solution for Divide and Conquer

- Suppose a complexity function T(n) is eventually non-decreasing and satisfies

$$T(n) = aT\left(\frac{n}{b}\right) + f(n) \qquad for\ n > 1,\ \ n\ a\ power\ of\ b$$

$$T(1) = d$$

- If $f(n) \in \Theta(n^k)$ Where $b >= 2$ and $k >= 0$ are constant integers and $a$, $c$, and d are constants such that $a > 0$ , $c > 0$ and $d >= 0$. Then

$$T(n) \in \begin{cases} \Theta(n^k) & if\ a < b^k \\ \Theta(n^k \lg n) & if\ a = b^k \\ \Theta(n^{\log_b a}) & if\ a > b^k \end{cases}$$

(This is slightly more general than the version in the book. Proof in Cormen et al. Intro to Algorithms 2nd edition.)

# Applying The Theorem

- Let's take another look at $t_n = 2t_{n/2} + n$.

- If we replace

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

- With

$$T(n) \leq aT\left(\frac{n}{b}\right) + f(n) \qquad or \qquad T(n) \geq aT\left(\frac{n}{b}\right) + f(n)$$

- Then the same result holds, replacing $\Theta$ with "big O" or $\Omega$ respectively.

# Homogeneous Linear Recurrences

- Another cookbook approach.
- Homogeneous Linear Recurrences have the form:

$$a_0 t_n + a_1 t_{n-1} + \cdots + a_k t_{n-k} = 0$$

- Where each $a_i$ is a constant.
- For example:
  - $7t_n - 3t_{n-1} = 0$
  - $6t_n - 5t_{n-1} + 8t_{n-2} = 0$
- Recall the Fibonacci sequence: $t_n = t_{n-1} + t_{n-2}$ , or

$$t_n - t_{n-1} - t_{n-2} = 0$$

# Example Solution

- Consider this recurrence:

$$t_n - 5t_{n-1} + 6t_{n-2} = 0$$

$$t_0 = 0$$

$$t_1 = 1$$

- Substitute $t_n = r^n$:

$$r^n - 5r^{n-1} + 6r^{n-2} = 0$$

- A little algebra:

$$r^{n-2}(r^2 - 5r + 6) = 0$$

- Factor:

$$r^{n-2}(r-2)(r-3) = 0$$

# Example Solution Continued

- So, we have roots at $r = 0$, $r = 2$ and $r = 3$.
- This means that $t_n = 0$, $t_n = 2^n$, $t_n = 3^n$ are all solutions to the recurrence.
- It turns out that the general solution can be specified as:

$$t_n = c_1 2^n + c_2 3^n$$

- Where $c_1$ and $c_2$ are arbitrary constants.
- We fix the constants by plugging in the initial conditions

# General Solution

Let the homogeneous linear recurrence equation

$$a_0 t_n + a_1 t_{n-1} + \cdots + a_k t_{n-k} = 0$$

be given. If its characteristic equation

$$a_0 r^k + a_1 r^{k-1} + \cdots + a_k r^0 = 0$$

has $k$ distinct solutions $r_1$, $r_2$, ... ,$r_k$, then the only solutions to the recurrence are:

$$t_n = c_1 r_1^n + c_2 r_2^n + \cdots + c_k r_k^n$$

The values of the constants are determined by the initial conditions

# Example (From Book)

$$t_n - 3t_{n-1} - 4t_{n-2} = 0$$

$$t_0 = 0$$

$$t_1 = 1$$

# What if there are Repeated Roots?

- Theorem B.2 in the book gives the technique for handling repeated roots:

- Let $r$ be a root of multiplicity $m$ of the characteristic equation for a homogeneous linear recurrence with constant coefficients. Then

$$t_n = r^n, \quad t_n = nr^n, \quad t_n = n^2 r^n, \quad \cdots \quad , t_n = n^{m-1} r^n$$

- are all solutions to the recurrence. Therefore a term for each is included in the general solution.

- E.g. if the characteristic equation had the solution:

$$(r-1)(r-3)^3$$

- The solution to the recurrence would be:

$$t_n = c_1 1^n + c_2 3^n + c_3 n 3^n + c_4 n^2 3^n$$

# Non-Homogeneous Linear Recurrences

$$a_0 t_n + a_1 t_{n-1} + \cdots + a_k t_{n-k} = f(n)$$

- There is no general method that works for any *f(n)*.
- We will show a cookbook method for $f(n) = b^n p(n)$.
- *b* is a constant and *p(n)* is any polynomial in *n*.
- Examples
  - $t_n + 5t_{n-1} - 6t_{n-2} = 4^n$ , here $b = 4$, $p(n) = 1$
  - $6t_n - 5t_{n-1} + 8t_{n-2} = 2^n(3n^2 + n + 1)$ , $b = 2$, $p(n) = 3n^2+n+1$

# General Solution

- A non-homogeneous linear recurrence of the form

$$a_0 t_n + a_1 t_{n-1} + \cdots + a_k t_{n-k} = b^n p(n)$$

- can be transformed into a homogeneous linear recurrence that has the characteristic equation

$$(a_0 r^k + a_1 r^{k-1} + \cdots + a_k)(r-b)^{d+1} = 0$$

- where *d* is the degree of *p(n)*

# Other Approaches...

- Change of Variables