Data Structures and Algorithms
Solving Recurrence Relations

Chris Brooks

Department of Computer Science
University of San Francisco
for (i=1; i<=n*n; i++)
    for (j=0; j<i; j++)
        sum++;
for (i=1; i<=n*n; i++) Executed n*n times
  for (j=0; j<i; j++) Executed <= n*n times
    sum++;
    O(1)

Running Time: $O(n^4)$

But can we get a tighter bound?
for (i=1; i<=n*n; i++)
    for (j=0; j<i; j++)
        sum++;

Exact # of times `sum++` is executed:

\[
\sum_{i=1}^{n^2} i = \frac{n^2(n^2 + 1)}{2}
\]

\[
= \frac{n^4 + n^2}{2}
\]

\[
\in \Theta(n^4)
\]
long power(long x, long n)
    if (n == 0)
        return 1;
    else
        return x * power(x, n-1);

How many times is this executed?
4-4: Recurrence Relations

\[ T(n) = \text{Time required to solve a problem of size } n \]

Recurrence relations are used to determine the running time of recursive programs – recurrence relations themselves are recursive

\[ T(0) = \text{time to solve problem of size 0} \]
\[ \quad - \text{Base Case} \]

\[ T(n) = \text{time to solve problem of size } n \]
\[ \quad - \text{Recursive Case} \]
4-5: Recurrence Relations

```c
long power(long x, long n)
    if (n == 0)
        return 1;
    else
        return x * power(x, n-1);
```

\[
T(0) = c_1 \quad \text{for some constant } c_1 \\
T(n) = c_2 + T(n - 1) \quad \text{for some constant } c_2
\]
4-6: Solving Recurrence Relations

\[ T(0) = c_1 \]
\[ T(n) = T(n - 1) + c_2 \]

If we knew \( T(n - 1) \), we could solve \( T(n) \).

\[ T(n) = T(n - 1) + c_2 \]
4-7: Solving Recurrence Relations

\[ T(0) = c_1 \]
\[ T(n) = T(n - 1) + c_2 \]

If we knew \( T(n - 1) \), we could solve \( T(n) \).

\[
T(n) \quad = T(n - 1) + c_2 \\
= T(n - 2) + c_2 + c_2 \\
= T(n - 2) + 2c_2
\]
4-8: Solving Recurrence Relations

\[ T(0) = c_1 \]
\[ T(n) = T(n - 1) + c_2 \]

If we knew \( T(n - 1) \), we could solve \( T(n) \).

\[
T(n) = T(n - 1) + c_2 \\
= T(n - 2) + c_2 + c_2 \\
= T(n - 2) + 2c_2 \\
= T(n - 3) + c_2 + 2c_2 \\
= T(n - 3) + 3c_2
\]
4-9: Solving Recurrence Relations

\[ T(0) = c_1 \]
\[ T(n) = T(n - 1) + c_2 \]

If we knew \( T(n - 1) \), we could solve \( T(n) \).

\[
\begin{align*}
T(n) & = T(n - 1) + c_2 \\
& = T(n - 2) + c_2 + c_2 \\
& = T(n - 2) + 2c_2 \\
& = T(n - 3) + c_2 + 2c_2 \\
& = T(n - 3) + 3c_2 \\
& = T(n - 4) + 4c_2 \\
\end{align*}
\]
4-10: Solving Recurrence Relations

\[ T(0) = c_1 \]
\[ T(n) = T(n - 1) + c_2 \]

If we knew \( T(n - 1) \), we could solve \( T(n) \).

\[ \begin{align*}
T(n) & = T(n - 1) + c_2 \\
& = T(n - 2) + c_2 + c_2 \\
& = T(n - 2) + 2c_2 \\
& = T(n - 3) + c_2 + 2c_2 \\
& = T(n - 3) + 3c_2 \\
& = T(n - 4) + 4c_2 \\
& = \ldots \\
& = T(n - k) + kc_2 
\end{align*} \]
$T(0) = c_1$

$T(n) = T(n - k) + k \times c_2$  \text{ for all } k

If we set $k = n$, we have:

$T(n) = T(n - n) + nc_2$

$= T(0) + nc_2$

$= c_1 + nc_2$

$\in \Theta(n)$
Can we avoid making a linear number of function calls?

long power(long x, long n)
    if (n==0) return 1;
    if (n==1) return x;
    if ((n % 2) == 0)
        return power(x*x, n/2);
    else
        return power(x*x, n/2) * x;
long power(long x, long n)
    if (n==0) return 1;
    if (n==1) return x;
    if ((n % 2) == 0)
        return power(x*x, n/2);
    else
        return power(x*x, n/2) * x;

\[ T(0) = c_1 \]
\[ T(1) = c_2 \]
\[ T(n) = T(n/2) + c_3 \]
(Assume \( n \) is a power of 2)
4-14: Solving Recurrence Relations

\[ T(n) = T(n/2) + c_3 \]
4-15: Solving Recurrence Relations

\[ T(n) = T(n/2) + c_3 \quad T(n/2) = T(n/4) + c_3 \]
\[ = T(n/4) + c_3 + c_3 \]
\[ = T(n/4) + 2c_3 \]
4-16: Solving Recurrence Relations

\[ T(n) = T(n/2) + c_3 \]
\[ = T(n/4) + c_3 + c_3 \]
\[ = T(n/4) + 2c_3 \]
\[ = T(n/8) + c_3 + 2c_3 \]
\[ = T(n/8) + 3c_3 \]

\[ T(n/2) = T(n/4) + c_3 \]
\[ = T(n/4) + c_3 + c_3 \]
\[ = T(n/4) + 2c_3 \]
\[ = T(n/8) + c_3 + 2c_3 \]
\[ = T(n/8) + 3c_3 \]
4-17: Solving Recurrence Relations

\[ T(n) = T(n/2) + c_3 \]
\[ = T(n/4) + c_3 + c_3 \]
\[ = T(n/4) + 2c_3 \]
\[ = T(n/8) + c_3 + 2c_3 \]
\[ = T(n/8) + 3c_3 \]
\[ = T(n/16) + c_3 + 3c_3 \]
\[ = T(n/16) + 4c_3 \]
\[ T(n/2) = T(n/4) + c_3 \]
\[ T(n/4) = T(n/8) + c_3 \]
\[ T(n/4) = T(n/8) + c_3 \]
\[ T(n/8) = T(n/16) + c_3 \]
4-18: Solving Recurrence Relations

\[ T(n) = T(n/2) + c_3 \]
\[ = T(n/4) + c_3 + c_3 \]
\[ = T(n/4)2c_3 \]
\[ = T(n/8) + c_3 + 2c_3 \]
\[ = T(n/8)3c_3 \]
\[ = T(n/16) + c_3 + 3c_3 \]
\[ = T(n/16) + 4c_3 \]
\[ = T(n/32) + c_3 + 4c_3 \]
\[ = T(n/32) + 5c_3 \]

\[ T(n/2) = T(n/4) + c_3 \]
\[ = T(n/4) = T(n/8) + c_3 \]
\[ = T(n/8) = T(n/16) + c_3 \]
\[ = T(n/16) = T(n/32) + c_3 \]
4-19: Solving Recurrence Relations

\[ T(n) = T(n/2) + c_3 \]
\[ = T(n/4) + c_3 + c_3 \]
\[ = T(n/4)2c_3 \]
\[ = T(n/8) + c_3 + 2c_3 \]
\[ = T(n/8)3c_3 \]
\[ = T(n/16) + c_3 + 3c_3 \]
\[ = T(n/16) + 4c_3 \]
\[ = T(n/32) + c_3 + 4c_3 \]
\[ = T(n/32) + 5c_3 \]
\[ = \ldots \]
\[ = T(n/2^k) + kc_3 \]
4-20: Solving Recurrence Relations

\[ T(0) = c_1 \]
\[ T(1) = c_2 \]
\[ T(n) = T(n/2) + c_3 \]
\[ T(n) = T(n/2^k) + kc_3 \]

We want to get rid of \( T(n/2^k) \). We get to a relation we can solve directly when we reach \( T(1) \)

\[ n/2^k = 1 \]
\[ n = 2^k \]
\[ \lg n = k \]
4-21: Solving Recurrence Relations

\[ T(0) = c_1 \]
\[ T(1) = c_2 \]
\[ T(n) = T(n/2) + c_3 \]

\[ T(n) = T(n/2^k) + kc_3 \]

We want to get rid of \( T(n/2^k) \). We get to a relation we can
solve directly when we reach \( T(1) \)
\[ \lg n = k \]

\[ T(n) = T(n/2^{\lg n}) + \lg n c_3 \]
\[ = T(1) + c_3 \lg n \]
\[ = c_2 + c_3 \lg n \]
\[ \in \Theta(\lg n) \]
long power(long x, long n)
    if (n==0) return 1;
    if (n==1) return x;
    if ((n % 2) == 0)
        return power(x*x, n/2);
    else
        return power(x*x, n/2) * x;
long power(long x, long n)  
  if (n==0) return 1;
  if (n==1) return x;
  if ((n % 2) == 0)  
    return power(power(x,2), n/2);
  else  
    return power(power(x,2), n/2) * x;

This version of power will not work. Why?
4-24: Power Modifications

long power(long x, long n)
  if (n==0) return 1;
  if (n==1) return x;
  if ((n % 2) == 0)
    return power(power(x,n/2), 2);
  else
    return power(power(x,n/2), 2) * x;

This version of power also will not work. Why?
long power(long x, long n)
    if (n==0) return 1;
    if (n==1) return x;
    if ((n % 2) == 0)
        return power(x,n/2) * power(x,n/2);
    else
        return power(x,n/2) * power(x,n/2) * x;

This version of power does work.

What is the recurrence relation that describes its running time?
long power(long x, long n)  
    if (n==0) return 1;
    if (n==1) return x;
    if ((n % 2) == 0)
        return power(x,n/2) * power(x,n/2);
    else
        return power(x,n.2) * power(x,n/2) * x;

\[
T(0) = c_1 \\
T(1) = c_2 \\
T(n) = T(n/2) + T(n/2) + c_3 \\
\quad = 2T(n/2) + c_3
\]

(Again, assume n is a power of 2)
4-27: Solving Recurrence Relations

\[ T(n) = 2T(n/2) + c_3 \]
\[ = 2[2T(n/4) + c_3]c_3 \]
\[ = 4T(n/4) + 3c_3 \]
\[ = 4[2T(n/8) + c_3] + 3c_3 \]
\[ = 8T(n/8) + 7c_3 \]
\[ = 8[2T(n/16) + c_3] + 7c_3 \]
\[ = 16T(n/16) + 15c_3 \]
\[ = 32T(n/32) + 31c_3 \]
\[ \cdots \]
\[ = 2^kT(n/2^k) + (2^k - 1)c_3 \]
Solving Recurrence Relations

\[ T(0) = c_1 \]
\[ T(1) = c_2 \]
\[ T(n) = 2^k T(n/2^k) + (2^k - 1)c_3 \]

Pick a value for \( k \) such that \( n/2^k = 1 \):

\[ n/2^k = 1 \]
\[ n = 2^k \]
\[ \log n = k \]
\[ T(n) = 2^{\log n} T(n/2^{\log n}) + (2^{\log n} - 1)c_3 \]
\[ = nT(n/n) + (n - 1)c_3 \]
\[ = nT(1) + (n - 1)c_3 \]
\[ = nc_2 + (n - 1)c_3 \]
\[ \in \Theta(n) \]