Deriving Recurrence Relations

To derive a recurrence relation for the running time of an algorithm:

- Figure out what “n”, the problem size, is.
- See what value of n is used as the base of the recursion. It will usually be a single value (e.g. \( n = 1 \)), but may be multiple values. Suppose it is \( n_0 \).
- Figure out what \( T(n_0) \) is. You can usually use “some constant \( c \)”, but sometimes a specific number will be needed.
- The general \( T(n) \) is usually a sum of various choices of \( T(m) \) (for the recursive calls), plus the sum of the other work done. Usually the recursive calls will be solving a subproblems of the same size \( f(n) \), giving a term “\( a \cdot T(f(n)) \)” in the recurrence relation.

Examples

```plaintext
procedure bugs(n)
    if n = 1 then do something
    else
        bugs(n - 1);
        bugs(n - 2);
        for i := 1 to n do
            something
```

```plaintext
procedure daffy(n)
    if n = 1 or n = 2 then do something
    else
        daffy(n - 1);
        for i := 1 to n do
            do something new
        daffy(n - 1);
```

\[
T(n) = \begin{cases} 
    c & \text{if } n = n_0 \\
    a \cdot T(f(n)) + g(n) & \text{otherwise}
\end{cases}
\]
procedure elmer(n)
  if n = 1 then do something
  else if n = 2 then do something else
  else
    for i := 1 to n do
      elmer(n - 1);
      do something different

\[
T(n) = \left\{ \begin{array}{ll}
\end{array} \right.
\]

procedure yosemite(n)
  if n = 1 then do something
  else
    for i := 1 to n - 1 do
      yosemite(i);
      do something completely different

\[
T(n) = \left\{ \begin{array}{ll}
\end{array} \right.
\]

Analysis of Multiplication

function multiply(y, z)
  comment return the product yz
  1. if z = 0 then return(0) else
  2. if z is odd
  3. then return(multiply(2y, [z/2]) + y)
  4. else return(multiply(2y, [z/2]))

Let \( T(n) \) be the running time of \( \text{multiply}(y, z) \),
where \( z \) is an \( n \)-bit natural number.

Then for some \( c, d \in \mathbb{R} \),
\[
T(n) = \left\{ \begin{array}{ll}
c & \text{if } n = 1 \\
T(n - 1) + d & \text{otherwise}
\end{array} \right.
\]
Solving Recurrence Relations

Use repeated substitution.

Given a recurrence relation $T(n)$.
- Substitute a few times until you see a pattern
- Write a formula in terms of $n$ and the number of substitutions $i$.
- Choose $i$ so that all references to $T()$ become references to the base case.
- Solve the resulting summation

This will not always work, but works most of the time in practice.

The Multiplication Example

We know that for all $n > 1$,

$$T(n) = T(n - 1) + d.$$

Therefore, for large enough $n$,

$$T(n) = T(n - 1) + d$$
$$T(n - 1) = T(n - 2) + d$$
$$T(n - 2) = T(n - 3) + d$$

$$\vdots$$

$$T(2) = T(1) + d$$
$$T(1) = c$$

Repeated Substitution

$$T(n) = T(n - 1) + d$$
$$= (T(n - 2) + d) + d$$
$$= T(n - 2) + 2d$$
$$= (T(n - 3) + d) + 2d$$
$$= T(n - 3) + 3d$$

There is a pattern developing. It looks like after $i$ substitutions,

$$T(n) = T(n - i) + id.$$

Now choose $i = n - 1$. Then

$$T(n) = T(1) + d(n - 1)$$
$$= dn + c - d.$$
Reality Check

We claim that

\[ T(n) = dn + c - d. \]

Proof by induction on \( n \). The hypothesis is true for \( n = 1 \), since \( d + c - d = c \).

Now suppose that the hypothesis is true for \( n \). We are required to prove that

\[ T(n + 1) = dn + c. \]

Now,

\[
\begin{align*}
T(n + 1) &= T(n) + d \\
&= (dn + c - d) + d \quad \text{(by ind. hyp.)} \\
&= dn + c.
\end{align*}
\]