University of Illinois at Urbana-Champaign
Department of Computer Science

Lecture 11: June 27

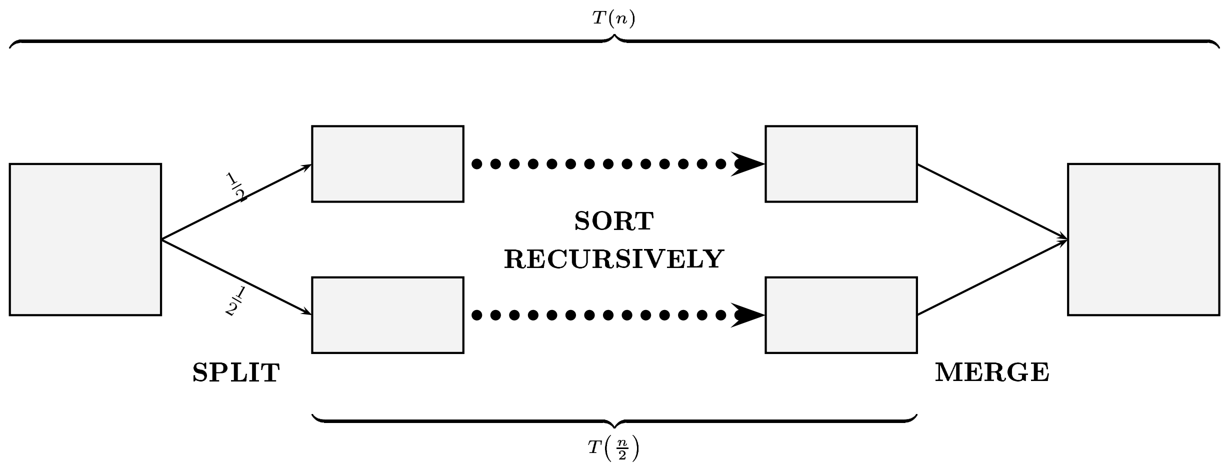
CS 273 Introduction to Theoretical Computer Science
Summer Semester, 2001

1 Domain and range transformations

Frank and Ernest
MATH WORKSHOP ← LEARN TO DO ARITHMETIC IN HALF THE TIME
ARE YOU INTERESTED, ERNIE?
THAT DEPENDS. IS HALF THE TIME GOOD OR BAD?
Copyright (c) 1992 by Thaves. Distributed from www.thecomics.com

1.1 An analysis of mergesort

A classic divide-and-conquer algorithm is mergesort. To sort an array, we divide it in half, sort each half recursively, and merge the two halves:



The running time of mergesort is given by the recurrence relation $T(n) = 2T(\frac{n}{2}) + kn$, where $T(1) = 1$.[1] How do we approach this? We have no techniques that allow us to work with such terms as $2T(\frac{n}{2})$. However, we can transform the recurrence into one we can work with.

[1]In fact, the correct recurrence is $T(n) = T(\lfloor \frac{n}{2} \rfloor) + T(\lceil \frac{n}{2} \rceil) + kn$, as n may be odd and T is only defined on integers. We ignore this point here, though, and derive an approximation.

Let $n = 2^i$. So $T(2^0) = 1$ and $T(2^i) = 2T(\frac{2^i}{2}) + k2^i = 2T(2^{i-1}) + k2^i$. Now, define the sequence t by $t_i = T(2^i)$. That gives us:

$$\begin{aligned} t_0 &= 1 \\ t_i &= 2t_{i-1} + k2^i \end{aligned}$$

The homogeneous part of this equation is annihilated by $\mathbf{E}-2$ and the non-homogeneous part of the equation is annihilated by $\mathbf{E}-2$. Thus $(\mathbf{E}-2)^2$ annihilates the entire equation, and we know that t_i is of the form $t_i = (ci + \hat{c})2^i$.

Before we declare success, we must transform the solution back to the vocabulary of the original problem:

$$\begin{aligned} t_i &= (ci + \hat{c})2^i \\ T(2^i) &= (ci + \hat{c})2^i \\ T(n) &= (c \lg n + \hat{c})n \\ &= cn \lg n + \hat{c}n \\ &= \Theta(n \lg n) \end{aligned}$$

We are using the technique of **domain and range transformations**.

1.2 A new algorithm to compute factorial

We have seen how factorial

$$n! = n(n-1)(n-2)\ldots 1$$

can be calculated by a recursive algorithm:

```
$factorial(n)$ {
    if $n = 0$ then return 1
    else return $n \times factorial(n-1)$
}
```

In analyzing this kind of problems, we pay more attention to the number of multiplications than the number of additions executed. The reason is that multiplications are often much more expensive than additions. If we count the number of bit operations, the cost of adding two numbers with k digits is $\theta(k)$, while the cost of multiplying them is $\theta(k^2)$. In analysis in below, we use $C(n)$ to denote the cost of calculating $n!$, which is the number of multiplications executed in the calculation.

Let's first look at the cost of the naive recursive algorithm given above. We have the below identities:

$$C(0) = 0$$

$$C(n) = C(n-1) + 1$$

Using what we learned from last lecture, the annihilator for this recursion is $(\mathbf{E}-1)^2$. Therefore the $C(n)$ should be of the form $an + b$. By plugging in the first two items we get

$$C(n) = n$$

Now let's look at a more subtle solution. The key idea is that half of the numbers in the range are even numbers. We can first get the products of all odd numbers in the range. Now if we take the factor 2 out of the even numbers, calculating the product of the resulting smaller numbers involves calculating the product of odd numbers in a smaller range, which is part of the work we have done. The multiplications in calculating the product of these odd numbers can be saved by doing an extra square operation. Of course we need to multiply those 2s we took out, but calculating power of 2 is a much easier job for the computer.

We have a set of algorithms:

Compute n squared:

```
$square(n)$ {
    return $n \times n$
}
```

Compute 2 raised to the nth power:

```
$poweroftwo(n)$ {
    if $n = 0$ return 1
    else {
        $x \leftarrow poweroftwo(\lfloor \frac{n}{2} \rfloor)$
        if $n$ is even
            return $square(x)$
        else
            return $square(x) \times 2$
    }
}
```

Compute the product of all odd integers in the range [a..b]:

```
$odds(a, b)$ {
    $nextodd \leftarrow 2 \times \lfloor \frac{a}{2} \rfloor + 1$
    if $b < nextodd$
        return 1
    else
        return $nextodd \times odds(a + 2, b)$
}
```

Compute n!:

```
$factorial(n)$ {
    if $n < 2$
        return 1
    else {
        $h \leftarrow \lfloor \frac{n}{2} \rfloor$
        $q \leftarrow \lfloor \frac{h}{2} \rfloor$
        return $square(odds(1, h)) \times odds(h + 1, n) \times factorial(q) \times poweroftwo(h + q)$
    }
}
```

It's easy to see that the number of multiplications performed in calculating $odds(1, h)$ and $odds(h+1, n)$ are both about $\frac{n}{4}$ (multiplying by 2 is just shifting one bit). $square()$ is simply one multiplication. The problem left is to figure out the cost of the $poweroftwo()$ operation.

For the cost of $poweroftwo()$, we have the below recurence relation:

$$C(0) = 0$$

$$C(k) = C(\frac{1}{2}k) + 2$$

This is what we call "secondary recurence". We will try to use domain transformation to solve it.

We will use two auxiliary sequences: a_n and b_n. a_n is defined as:

$$a_n = 2 \times a_{n-1}$$

$$a_i = k$$

$$a_0 = 1$$

Solving this recurence we get

$$a_n = 2^n$$

and b_n is defined as:

$$b_n = C(a_n)$$

Then we have

$$b_n = b_{n-1} + 2$$

We know how to solve it! The annililator is $(\mathbf{E} - 1)^2$ and b_i should be of the form $ci + \hat{c}$. So we have

$$C(a_i) = ci + \hat{c}$$

Since $a_i = k$, $i = \lg k$ and there is

$$C(k) = c \lg k + \hat{c}$$

Plugging in $C(1) = 2$ and $C(2) = 4$ we get the solution

$$C(k) = 2 \lg k + 2$$

Therefore the cost of $poweroftwo()$ part in the $factorial()$ algorithm is $2 \lg\left(\frac{3}{4}n\right)$. We can ignore the constant and simplify that to $2 \lg n$.

Now we can apply the same method to the cost of $factorial$. We have a recurence relation like this:

$$C(k) = C\left(\frac{k}{4}\right) + \frac{k}{2} + 2 \lg k$$

$$C(0) = C(1) = 1$$

Again we define a_n and b_n similarly. Here

$$a_n = 4^n$$

$$a_i = k$$

$$b_n = C(a_n)$$

Then we have

$$b_i = b_{i-1} + \frac{4^i}{2} + 2i$$

The annihilator for b is $(\mathbf{E} - 1)^3(\mathbf{E} - 4)$. Therefore

$$C(a_i) = (ui^2 + vi + w)1^i + y4^i$$

Since $4^i = k$, $i = \frac{\lg k}{2}$,

$$C(k) = u\left(\frac{\lg k}{2}\right)^2 + v\left(\frac{\lg k}{2}\right) + w + yk$$

Plugging in the first a few items we can get

$$C(k) = \frac{2}{3}k + O(\lg^2 k)$$

Comparing this to the naive recursive alaorithm we see that we are saving about $\frac{2}{3}$ of the multiplications. By furthering the algorithm we can achieve a performance of $O(\frac{1}{2}k)$.

1.3 Another example

Consider the recurrence $T(n) = 4T(\frac{n}{2}) + kn$, where $T(1) = 1$. Let $n = 2^i$, as in the earlier example, and define $t_i = T(2^i)$. We have:

$$\begin{aligned} t_0 &= 1 \\ t_i &= 4t_{i-1} + k2^i \end{aligned}$$

This is annihilated by $(\mathbf{E} - 2)(\mathbf{E} - 4)$, so t_i is of the form $t_i = c2^i + \hat{c}4^i$. Restoring our original notation:

$$\begin{aligned} t_i &= c2^i + \hat{c}4^i \\ T(2^i) &= c2^i + \hat{c}4^i \\ T(n) &= cn + \hat{c}n^2 \\ &= \Theta(n^2) \end{aligned}$$

1.4 A challenging example

Consider $T(n) = 2T(\frac{n}{3} - 1) + n$. Define u:

$$\begin{aligned} u_i &= n \\ u_{i-1} &= \frac{n}{3} - 1 \end{aligned}$$

From this we arrive at the **secondary recurrence** $u_i = 3u_{i-1} + 3$, which is annihilated by $(\mathbf{E} - 1)(\mathbf{E} - 3)$. Thus $u_i = c3^i + \hat{c}$.

Substituting into $T(n)$, we obtain $T(u_i) = 2T(u_{i-1}) + u_i$. Letting $t_i = T(u_i)$, we arrive at the recurrence $t_i = 2t_{i-1} + c3^i + \hat{c}$, which by now we are able to solve.