Analysis of Divide-and-Conquer Algorithms

The divide-and-conquer paradigm (Ch 2.3)
- divide **the problem into a number of subproblems**
- conquer **the subproblems by solving them**
- combine **the subproblem solutions to get the final solution**

add all these to get recurrence relation for T(n)

Example: Merge-Sort
- divide **the n-element input sequence to be sorted into two n/2-element subsequences.**
- conquer **the subproblems recursively using merge sort.**
- combine **the resulting two sorted n/2-element sequences by merging.**

Analyzing Divide-and-Conquer Algorithms

A recursive algorithm can often be described by a *recurrence equation* that describes the overall runtime on a problem of size *n* in terms of the runtime on smaller inputs.

For divide-and-conquer algorithms, we get recurrences like:

$$T(n) = \begin{cases} 1 & \text{if } n \le c \\ aT(n/b) + D(n) + C(n) & \text{otherwise} \end{cases}$$

where

- a = number of subproblems we divide the problem into
- n/b = size of the subproblems (in terms of n)
- $D(n)$ = time to divide the size n problem into subproblems
- $C(n)$ = time to combine the subproblem solutions to get the answer for the problem of size n

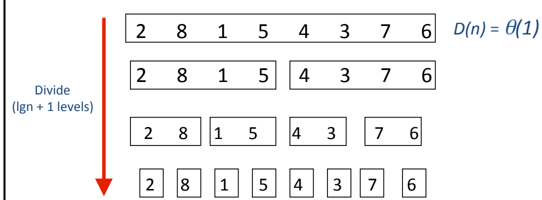
```
Merge-Sort(A,p,r)
if p < r then
    q = ⌊(p+r)/2⌋
    Merge-Sort(A,p,q)
    Merge-Sort(A,q+1,r)
    Merge(A,p,q,r)
```

Initial call:
`Merge-sort(A,1,length(A))`

The Merge subroutine takes linear time to merge n elements that are divided into two *sorted* arrays of $n/2$ elements each.

```
Merge(A,p,q,r)
n1 = q-p+1; n2 = r-q;
Create arrays
 L[1...n1+1] and
 R[1...n2+1]
for i = 1 to n1
    L[i] = A[p+i-1]
for i = 1 to n2
    R[i]= A[q+i]
L[n1+1] = R[n2+1] = ∞
i = j = 1
for k = p to r
  if L[i] ≤ R[j]
     A[k] = L[i]
     i = i+1
  else A[k] = R[j]
     j = j+1
```

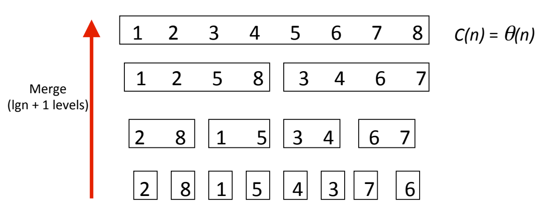
Analyzing Merge-Sort



Why are there lgn + 1 levels?

How long does it take to find the midpoint of an array?

Analyzing Merge-Sort



$$T(n) = \begin{cases} \theta(1) & \text{if } n = 1 \\ 2T(n/2) + \theta(n) & \text{otherwise} \end{cases}$$

Recurrence for worst-case running time for Merge-Sort

Analyzing Merge-Sort

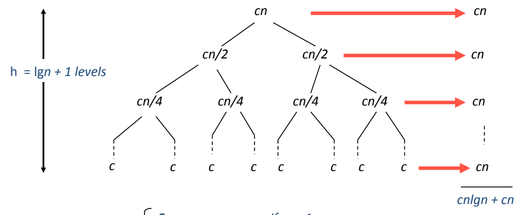
$$T(n) = \begin{cases} \theta(1) & \text{if } n = 1 \\ 2T(n/2) + \theta(n) & \text{otherwise} \end{cases}$$

Recurrence for worst-case running time for Merge-Sort

$$aT(n/b) + D(n) + C(n)$$

- a = 2 (two subproblems)
- n/b = n/2 (each subproblem has size approx $n/2$)
- $D(n)$ = $\Theta(1)$ (just compute midpoint of array)
- $C(n)$ = $\Theta(n)$ (merging can be done by scanning sorted subarrays)

Recurrence Tree for Merge-Sort



$$T(n) = \begin{cases} c & \text{if } n = 1 \\ 2T(n/2) + cn & \text{otherwise} \end{cases}$$

Recurrence tree for Merge-Sort

Review of Logarithms

A logarithm is an inverse exponential function. Saying $b^x = y$ is equivalent to saying $\log_b y = x$.

- **notation convention for logarithms:**
  - **lg*n* = $\log_2 n$ *(binary logarithm -- note, no subscript, just lg)***
  - ln*n* = $\log_e n$ *(natural logarithm)*
- **properties of logarithms:**
  - $\log_b(xy) = \log_b x + \log_b y$
  - $\log_b (x/y) = \log_b x - \log_b y$
  - $\log_b x^a = a\log_b x$
  - $\log_b a = \log_x a / \log_x b$
  - $a = b^{\log_b a}$ (e.g., $n = 2^{\lg n} = n^{\lg 2}$)

Log functions grow very slowly as n grows without bound.

General Plan for calculating running time of recursive algorithms

1. Decide on a parameter indicating input size.
2. Set up a recurrence relation, with the appropriate base cases.
3. Solve the recurrence or otherwise ascertain the order of growth using, e.g. backward substitution, the master method, a recursion tree, or a good guess.

Solving Recurrences

I will cover the first 2 techniques to solve recurrences. The third method is covered in the book (as is solving with a good guess).

1. Backward Substitution: involves substituting next step into equation until you see a pattern, converting the pattern to a summation, and solving the summation.
2. Apply the "Master Theorem": If the recurrence has the form
   $$T(n) = aT(n/b) + f(n)$$
   then there is a formula that can (often) be applied, given in § 4-5.
3. Apply the recursion tree method from § 4-4.

To make the solutions simpler, we will

- assume base cases are constant time, i.e., $T(n) = \Theta(1)$ for small enough *n*.

Solving recurrence for n!

```
Algorithm F(n)
  Input:  a positive integer n
  Output:  n!
1.  if n=1
2.       return 1
3.  else
4.       return F(n-1) * n
```

We can solve this recurrence (ie, find an expression of the running time T(n) that is not given in terms of itself) using a method known as *backward substitution.*

T(n) for the factorial problem

For recursive algorithms such as computing the *factorial* of n, we get an expression like the following:

$$\mathbf{T(n)} = \begin{cases} 1 & \text{if } n = 0 \\ T(n-1) + D(n) + C(n) & \text{otherwise} \end{cases}$$

where

- *n-1* = size of the subproblem (in terms of *n*)
- *D(n)* = time to divide the size *n* problem into subproblems
- *C(n)* = time to combine the subproblem solutions to get the answer for the problem of size *n*

Solving recurrence for n!

Algorithm F(n)
  Input: a positive integer n
  Output: n!
1. **if** n=1
2.   **return** 1
3. **else**
4.   **return** F(n-1) * n

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

$$
\begin{aligned}
T(n) &= T(n-1) + 1 \quad (\text{ subst } T(n-1) = [T(n-2) + 1]\ ) \\
&= [T(n-2) + 1] + 1 = T(n-2) + 2 \\
&\qquad (\text{ subst } T(n-2) = [T(n-3) + 1]\ ) \\
&= [T(n-3) + 1] + 2 = T(n-3) + 3 \ldots \\
&\ldots \\
&= T(n-i)+i = \ldots = T(n-n)+n = 0 + n
\end{aligned}
$$

Therefore, this algorithm has linear running time.

We can solve this recurrence (ie, find an expression of the running time T(n) that is not given in terms of itself) using a method known as *backward substitution.*

Solving Recurrences: Back Substitution

Example: $T(n) = 2T(n/2) + n$ (look familiar?)

$$
\begin{aligned}
T(n) &= 2T(n/2) + n \\
&= 2[2T(n/4) + n/2] + n && \text{/* expand } T(n/2) \text{ */} \\
&= 4T(n/4) + n + n && \text{/* simplify */} \\
&= 4[2T(n/8) + n/4] + n + n && \text{/* expand } T(n/4) \text{ */} \\
&= 8T(n/8) + n + n + n && \text{/* simplify */} \\
&= \ldots && \ldots \\
&= 2^{\lg n}T(n/2^{\lg n}) + \ldots + n + n + n && \text{/* after lgn iterations */} \\
&= c2^{\lg n} + n\lg n \\
&= cn + n\lg n && \text{/* } 2^{\lg n} = n^{\lg 2} = n \text{ */} \\
&= \theta(n\lg n)
\end{aligned}
$$

Solving Recurrences: Back Substitution

Example: $T(n) = 4T(n/2) + n$

$$
\begin{aligned}
T(n) &= 4T(n/2) + n \\
&= 4[4T(n/4) + n/2] + n && \text{/* expand } T(n/2) \text{ */} \\
&= 16T(n/4) + 2n + n && \text{/* simplify } 16 = 4^2 \text{*/} \\
&= 16[4T(n/8) + n/4] + 2n + n && \text{/* expand } T(n/4) \text{ */} \\
&= 64T(n/8) + 4n + 2n + n && \text{/* simplify } 64 = 4^3 \text{*/} \\
&= \ldots && \ldots \\
&= 4^{\lg n}T(n/2^{\lg n}) + \ldots + 4n + 2n + n && \text{/* after lgn iterations */} \\
&= c4^{\lg n} + n \sum_{k=0}^{\lg n-1} 2^k = 2^0 + 2^1 + \ldots + 2^{\lg n-1} && \text{/* convert to summation */} \\
&= cn^{\lg 4} + n\,(2^{\lg n} - 1) && \text{/* } 4^{\lg n} = n^{\lg 4} = n^2 \text{ */} \\
&= cn^2 + n(n - 1) && \text{/* } 2^{\lg n} = n^{\lg 2} = n \text{ */} \\
&= \theta(n^2)
\end{aligned}
$$

Solving Recurrences: Back Substitution

Example: $T(n) = T(n/2) + 1$

$$
\begin{aligned}
T(n) &= T(n/2) + 1 \\
&= [T(n/4) + 1] + 1 && \text{/* expand } T(n/2) \text{ */} \\
&= T(n/4) + 2 && \text{/* simplify */} \\
&= [T(n/8) + 1] + 2 && \text{/* expand } T(n/4) \text{ */} \\
&= T(n/8) + 3 && \text{/* simplify */} \\
&= \ldots && \ldots \\
&= T(n/2^{\lg n}) + \lg n && \text{/* } 2^{\lg n} = n^{\lg 2} = n \text{ */} \\
&= c + \lg n \\
&= \theta(\lg n)
\end{aligned}
$$

Which well-known algorithm has this running time??

Solving Recurrences: Master Method (§4.5)

The master method provides a 'cookbook' method for solving recurrences where n is divided repeatedly by a constant. This is the method we will use most often for solving recurrences of the form

$$T(n) = aT(n/b) + f(n)$$

Where a is the number of sub-problems, n/b is the size of each subproblem, and f(n) is the time to divide or combine data.

Solving Recurrences: Master Method (§4.5)

Master Theorem: Let $a \geq 1$ and $b > 1$ be constants, let f(n) be a function, and let T(n) be defined on nonnegative integers as:

$$T(n) = aT(n/b) + f(n)$$

Where a is the number of sub-problems, n/b is the size of each sub-problem, and f(n) is the time to divide or combine data.

Then, T(n) can be bounded asymptotically as follows:

1. $T(n) = \theta(n^{\log_b a})$ if $f(n) \leq n^{\log_b a-\varepsilon}$ for some constant $\varepsilon > 0$
2. $T(n) = \theta(n^{\log_b a}\lg n)$ if $f(n) = n^{\log_b a}$
3. $T(n) = \theta f(n)$ if $f(n) \geq n^{\log_b a+\varepsilon}$ for some constant $\varepsilon > 0$

Alternate Version of Master Method

Let $a \geq 1$, $b > 1$, $k \geq 0$ be constants, let p be a real number, and let T(n) be defined on nonnegative integers as:

$$T(n) = aT(n/b) + \theta(n^k \log^p n)$$

Then, T(n) can be bounded asymptotically as follows:

1. If $a > b^k$, then $T(n) = \theta(n^{\log_b a})$
2. If $a = b^k$, then
   a) If $p > -1$, then $T(n) = \theta(n^{\log_b a} \log^{p+1} n)$
   b) If $p = -1$, then $T(n) = \theta(n^{\log_b a} \log\log n)$
   c) If $p < -1$, then $T(n) = \theta(n^{\log_b a})$
3. If $a < b^k$, then
   a) If $p \geq 0$, then $T(n) = \theta(n^k \log^p n)$
   b) If $p < 0$, then $T(n) = \theta(n^k)$

Alternate Version of Master Method

Let $a \geq 1$, $b > 1$, $k \geq 0$ be constants, let p be a real number, and let T(n) be defined on nonnegative integers as:

$$T(n) = aT(n/b) + \theta(n^k \log^p n)$$

Then, T(n) can be bounded asymptotically as follows:

1. If $a > b^k$, then $T(n) = \theta(n^{\log_b a})$
2. If $a = b^k$, then $T(n) = \theta(n^{\log_b a} \log^{p+1} n)$
3. If $a < b^k$, then $T(n) = \theta(n^k \log^p n)$

Solving Recurrences: Master Method

Example: $T(n) = 9T(n/3) + n$

Example: $T(n) = T(n/2) + 1$

Solving Recurrences: Master Method

Example: $T(n) = T(n/2) + n^2$

Example: $T(n) = 4T(n/2) + n^2$

Solving Recurrences: Master Method

Example: $T(n) = 7T(n/3) + n^2$

Example: $T(n) = 7T(n/2) + n^2$

Solving Recurrences: Alt. Master Method

Example: $T(n) = 9T(n/3) + n$

Example: $T(n) = T(n/2) + 1$

Solving Recurrences: Alt. Master Method

Example: $T(n) = T(n/2) + n^2$

Example: $T(n) = 4T(n/2) + n^2$

Solving Recurrences: Alt. Master Method

Example: $T(n) = 7T(n/3) + n^2$

Example: $T(n) = 7T(n/2) + n^2$