**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:

\[
T(n) = \begin{cases} 
1 & \text{if } n \leq c \\
\alpha T(n/b) + D(n) + C(n) & \text{otherwise}
\end{cases}
\]

- \( \alpha \) = 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 \( n \)

---

**Analyzing Merge-Sort**

- **Merge-Sort(A, p, r)**
  1. if \( p < r \) then
  2. \( q = \lfloor (p+r)/2 \rfloor \)
  3. Merge-Sort(A, p, q)
  4. Merge-Sort(A, q+1, r)
  5. 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.

---

**Analyzing Merge-Sort**

\[
C(n) = \Theta(n)
\]

\[
T(n) = \begin{cases} 
\Theta(1) & \text{if } n = 1 \\
2T(n/2) + \Theta(n) & \text{otherwise}
\end{cases}
\]

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)

---

**Analyzing Merge-Sort**

\[
\text{Divide (ign + 1 levels)}
\]

Why are there \( \lceil \log n \rceil + 1 \) levels?

How long does it take to find the midpoint of an array?
Recurrence Tree for Merge-Sort

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:**
  - $\log_n n = \log_2 n$ (binary logarithm – note, no subscript, just log)
  - $\ln n = \log_e n$ (natural logarithm)

- **properties of logarithms:**
  - $\log_b (xy) = \log_b x + \log_b y$
  - $\log_b \left(\frac{x}{y}\right) = \log_b x - \log_b y$
  - $\log_b a = \frac{\log_c a}{\log_c b}$
  - $a = b^{\log_b a}$ (e.g., $n = 2^{\log_2 n}$)

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 $F(n-1) \times n$

We can solve this recurrence (i.e., 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:

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

where
- $n-1 = \text{size of the subproblem (in terms of } n)$
- $D(n) = \text{time to divide the size } n \text{ problem into subproblems}$
- $C(n) = \text{time to combine the subproblem solutions to get the answer for the problem of size } n$
Solving recurrence for n!

Algorithm F(n)

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

T(n) = T(n-1) + 1 (substitute T(n-1) = T(n-2) + 1)
= [T(n-2) + 1] + 1 = T(n-2) + 2
= [T(n-3) + 1] + 1 = T(n-3) + 2...
= [T(n-k) + 1] + 1 = T(n-k) + 2
Therefore, this algorithm has linear running time.

We can solve this recurrence (i.e., 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) = 4T(n/2) + n

T(n) = 4T(n/2) + n
= 4[4T(n/4) + n/2] + n  /* expand T(n/2) */
= 16[4T(n/8) + n/4] + 2n + n  /* simplify */
= 64[4T(n/16) + n/8] + 4n + 2n + n
= ...  
= 4^kT(n/2^k) + ... + 4n + 2n + n  /* after log n iterations */
= 2^{log_2 n} + n log_2 n  /* convert to summation */
= 2n log_2 n, log_2 n = n */
= 0(n log_2 n)

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: Back Substitution

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

T(n) = 2T(n/2) + n
= 2[2T(n/4) + n/2] + n  /* expand T(n/2) */
= 4T(n/4) + n/2 + n  /* expand */
= 8T(n/8) + n + n + n  /* expand */
= ...  
= 2^kT(n/2^k) + ... + n + n + n  /* after log n iterations */
= cn log_2 n + n log_2 n
= 0(n log_2 n)

Which well-known algorithm has this running time??

Solving Recurrences: Master Method (§4.5)

Master Theorem: Let a ≥ 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 subproblem, and f(n) is the time to divide or combine data.

Then, T(n) can be bounded asymptotically as follows:
1. T(n) = 0(n log_b a) if f(n) ≤ n log_b a for some constant k > 0
2. T(n) = 0(n log_b a log n) if f(n) = n log_b a
3. T(n) = 0(n) if f(n) ≥ n log_b a for some constant k > 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 $a > b^p$, then $T(n) = \Theta(n^p)$
2. If $a = b^p$, then
   a) if $p > -1$, then $T(n) = \Theta(n^p \log^p n)$
   b) if $p = -1$, then $T(n) = \Theta(n^p \log \log n)$
   c) if $p < -1$, then $T(n) = \Theta(n^p)$
3. If $a < b^p$, then
   a) if $p \geq 0$, then $T(n) = \Theta(n^p \log^k n)$
   b) if $p < 0$, then $T(n) = \Theta(n^p)$

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

Solving Recurrences: Master Method

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

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

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

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

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: \( T(n) = 7 \cdot T\left(\frac{n}{2}\right) + n^2 \)

Example: \( T(n) = 4 \cdot T\left(\frac{n}{2}\right) + n^2 \)

Example: \( T(n) = 7 \cdot T\left(\frac{n}{3}\right) + n^2 \)

Example: \( T(n) = 7 \cdot T\left(\frac{n}{2}\right) + n^2 \)