Analysis of InsertionSort

InsertionSort sorts an array of elements in ascending order.

\[
\text{InsertionSort}(A) \quad \text{times} \quad 1.
\text{for } j = 2 \text{ to } \text{length}(A) \\
\quad 2. \quad \text{key} = A[j] \\
\quad 3. \quad i = j - 1 \\
\quad 4. \quad \text{while } i > 0 \text{ and } A[i] > \text{key} \\
\quad \quad 5. \quad A[i + 1] = A[i] \\
\quad \quad 6. \quad i = i - 1 \\
\quad 7. \quad A[i + 1] = \text{key}
\]

Analyzing Recursive Algorithms (Ch. 4)

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:

\[
T(n) = \begin{cases} 
\Theta(1) & \text{if } n \leq c \\
\beta T(n) + f(n) + \Theta(n) & \text{otherwise}
\end{cases}
\]

where

- \( \alpha = \) number of subproblems we divide the problem into
- \( \beta = \) 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 \( n \)

Solving Recurrences

We will use the following methods to solve recurrences

1. Backward Substitution.
2. Apply the "Master Theorem": If the recurrence has the form

\[
T(n) = \alpha T(n/b) + f(n)
\]

then there are 2 formulae that can (often) be applied; one of these is given in § 4-3.

Recurrence trees can be used along with backward substitution to guess the running time of a recurrence relation. Most recurrences of the form

\[
T(n) = \alpha T(n/b) + f(n)
\]

will be solved using the Master Theorem.

To make the solutions simpler, we will

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

Solving Recurrence with backward substitution

Algorithm F(n)

Input: a positive integer \( n \)

Output: \( n! \)

1. \text{if } n = 0 \\
   \quad \text{return } 1 \\
2. \quad \text{else} \\
3. \quad \text{return } F(n-1) \times n

\[
T(n) = T(n-1) + 1 \\
\quad \text{sub} T(n-1) = T(n-2) + 1 \\
\quad = T(n-2) + 1 + 1 = T(n-2) + 2 \\
\quad \text{sub} T(n-2) = T(n-3) + 1 \\
\quad = T(n-3) + 1 + 2 = T(n-3) + 3 \\
\cdots \\
\quad \text{sub} T(0) + 1 = T(0) + 1 + 1 \\
\quad = T(0) + n = T(0) + n = 0 + n = C(n)
\]

Therefore, this algorithm has linear running time.

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

Analyzing Recursive Algorithms

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

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

where

- \( \alpha = \) number of subproblems (1)
- \( n-1 = \) size of the subproblems (in terms of \( n \))
- \( D(n) = \) time to divide the size \( n \) problem into subproblems = 1
- \( C(n) = \) time to combine the subproblem solutions to get the answer for the problem of size \( n = 1 \)
**Binary Search (recursive version)**

Algorithm Binary-Search-Rec(A[1…n], k, l, r)

Input: a sorted array A of n comparable items, search key k, leftmost and rightmost index positions in A

Output: Index of array’s element that is equal to k or -1 if k not found

1. if (l > r) return -1; k not found
2. else
3. m = (l + r)/2; m is midpoint
4. if k = A[m]
5. return m; found index of k at A[m]
6. else if k < A[m]
7. return Binary-Search-Rec(A, k, l, m-1); look in lower half
8. else
9. return Binary-Search-Rec(A, k, m+1, r); look in upper half

What is the running time of this algorithm for an input of size n? We need to figure out what T(n) is for this algorithm.
Analysis of Divide-and-Conquer Algorithms

The divide-and-conquer paradigm (Ch.2)

• divide the problem into a number of subproblems
• conquer the subproblems (solve them)
• combine the subproblem solutions to get the solution to the original problem

Example: Merge Sort

• divide the n-element sequence to be sorted into two n/2-element sequences.
• conquer the subproblems recursively using merge sort.
• combine the resulting two sorted n/2-element sequences by merging.

```plaintext
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)

Merge(A,p,q,r)
1. n1 ← q-p+1; n2 ← r-q
2. Create arrays L[1...n1+1] and R[1...n2+1]
3. for i ← 1 to n1
5. for i ← 1 to n2
6. R[i] ← A[q+i]
7. L[n1+1] = R[n2+1] = ∞
8. i ← j ← 1
9. for k ← p to r
10. if L[i] ≤ R[j]
11. A[k] ← L[i]
12. i ← i+1
13. else A[k] ← R[j]
14. j ← j+1
```

Divide (log levels)

2 8 1 5 4 3 7 6

\(D(n) = \Theta(1)\)

Merge (log levels)

1 2 3 4 5 6 7 8

\(C(n) = \Theta(n)\)

Recurrence for worst-case running time for Merge-Sort

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

Recursion Tree for 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 of Merge-Sort
Solving Recurrences: Master Method (§4.3)

The master method provides a 'cookbook' method for solving recurrences of a certain form.

**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 subproblems, $n/b$ 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. $T(n) = \Theta(n^p)$ if $f(n) = O(n^{p+\varepsilon})$ for some constant $\varepsilon > 0$
2. $T(n) = \Theta(n \log^k n)$ if $f(n) = O(n \log^{k+\varepsilon} n)$
3. $T(n) = \Theta(f(n))$ if $f(n) = \Omega(n^{\log_b a + \varepsilon})$ 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:

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

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 $T(n) = \Theta(n^{p+\varepsilon} \log^{k+\varepsilon} n)$
   b) If $p = -1$, then $T(n) = \Theta(n \log^k n)$
   c) If $p < -1$, then $T(n) = \Theta(n)$
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 $T(n) = \Theta(n^p)$

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

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

**Solving Recurrences: Alt. Master Method**

**Example 1a:** $T(n) = 9T(n/3) + n$

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

**Master Method Restated**

**Master Theorem:** If $T(n) = aT(n/b) + O(n^d)$ for some constants $a \geq 1$, $b > 1$, $d \geq 0$, then

$$T(n) =
\begin{cases}
\Theta(n^p) & \text{if } d < \log_b a \quad (a > b^p) \\
\Theta(n \log^k n) & \text{if } d = \log_b a \quad (a = b^p) \\
\Theta(n^d) & \text{if } d > \log_b a \quad (a < b^p)
\end{cases}$$

*Why? The proof uses a recursion tree argument (given in our textbook).*

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

**Example:** $T(n) = T(n/2) + O(n)$

**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: Alt. Master Method

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

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

Example: \( T(n) = 7T(n/2) + n \)

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

Solving Recurrences: Master Method

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

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