Analysis of InsertionSort

InsertionSort sorts an array of elements in ascending order.

```
InsertionSort(A)                    times
for j = 2 to length[A]
      key = A[j]
      i = j - 1
      while i > 0 and A[i] > key

            A[i + 1] = A[i]

            i = i - 1
      A[i + 1] = key
```

Analysis of InsertionSort

```
InsertionSort(A)
for j = 2 to length[A]
    key = A[j]
    i = j - 1
    while i > 0 and A[i] > key
        A[i + 1] = A[i]
        i = i - 1
    A[i + 1] = key
```

- For insertion sort, does the running time vary for different input instances?

  If so, give an instance of best-case and worst-case inputs.

What is the raw running time for the best case?

$c_1n + c_2(n-1) + c_3(n-1) + c_4(n-1) + c_7(n-1)$

$= (c_1 + c_2 + c_3 + c_4 + c_7)n - (c_2 + c_3 + c_4 + c_7)$ = **an** - b

What is the raw running time for the worst case?

Add up terms on last slide.

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:

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

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

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

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

   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 $T(n) = aT(n/b) + f(n)$ 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!
if n=0
    return 1
else
    return F(n-1) * n

$T(n) = T(n-1) + 1$
$T(0) = 0$

$T(n) = T(n-1) + 1$ subst $T(n-1) = T(n-2) + 1$
$= [T(n-2) + 1] + 1 = T(n-2) + 2$
  subst $T(n-2) = T(n-3) + 1$
$= [T(n-3) + 1] + 2 = T(n-3) + 3$
…
$= T(n-i) + i =$
…
$= T(n-n) + n = T(0) + n = 0 + n = O(n)$

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:

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

where

- a = 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

Solving Recurrences: Backward Substitution

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

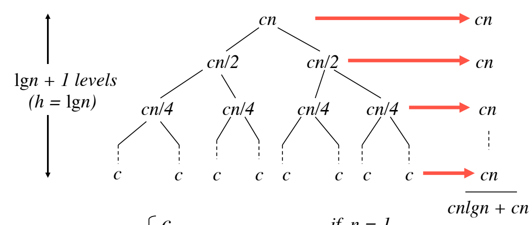
$T(n) = 2T(n/2) + n$
$= 2[2T(n/4) + n/2] + n$ /* expand T(n/2) */
$= 4T(n/4) + n + n$ /* simplify */
$= 4[2T(n/8) + n/4] + n + n$ /* expand T(n/4) */
$= 8T(n/8) + n + n + n$ /* simplify…see a pattern? */

. . . continue until $T(n/n) = T(1)$ is reached

$= 2^{\lg n}T(n/2^{\lg n}) + \ldots + n + n + n$ /* after lgn iterations */
$= 2^{\lg n}T(1) + \ldots + n + n + n$ /* $2^{\lg n} = n^{\lg 2} = n$ */
$= c2^{\lg n} + n\lg n$

$= cn + n\lg n$
$= O(n\lg n)$

Recursion Tree for T(n) = 2T(n/2) + n



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

Solving Recurrences: Backward Substitution

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

$T(n) = 2T(n/2) + 4n$
$= 2[2T(n/4) + 4(n/2)] + 4n$ /* expand T(n/2) */
$= 4T(n/4) + 8n/2 + 4n$ /* simplify */
$= 4T(n/4) + 4n + 4n$
$= 4[2T(n/8) + 4(n/4)] + 4n + 4n$ /* expand T(n/4) */
$= 8T(n/8) + 16n/4 + 4n + 4n$ /* simplify…see a pattern? */
$= 8T(n/8) + 4n + 4n + 4n$
. . . continue until $T(n/n) = T(1)$ is reached
$= 2^{\lg n}T(n/2^{\lg n}) + \ldots + 4n + 4n + 4n$ /* after lgn iterations */
$= nT(1) + \lg n(4n)$ /* $2^{\lg n} = n^{\lg 2} = n$ */

$= cn + 4n\lg n$

$= O(n\lg n)$

Solving Recurrences: Backward 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) */
$= 16T(n/4) + 4n/2 + n$ /* simplify */
$= 16[4T(n/8) + n/4] + 2n + n$ /* expand T(n/4) */
$= 64T(n/8) + 16n/4 + 2n + n$ /* simplify */
$= 64T(n/8) + 4n + 2n + n$
. . . continue until $T(n/n) = T(1)$ is reached
$= 4^{\lg n}T(n/2^{\lg n}) + \ldots + 4n + 2n + n$ /* after lgn iterations */

$= c4^{\lg n} + n\sum_{k=0}^{\lg n-1} 2^k = 2^0 + 2^1 + \ldots + 2^{\lg n-1}$ /* convert to summation */
/* p. 1147 */
$= cn^{\lg 4} + n(2^{\lg n} - 1)$ /* $4^{\lg n} = n^{\lg 4} = n^2$ */
$= cn^2 + n(n - 1)$ /* $2^{\lg n} = n^{\lg 2} = n$ */
$= O(n^2)$

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
if (l > r) return -1            ; k not found
else
   m = ⌊(l + r)/2⌋              ; m is midpoint
   if k = A[m]
      return m                  ; found index of k at A[m]
   else if k < A[m]
      return Binary-Search-Rec(A, k, l, m-1)   ; look in lower half
   else
      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.

Solving Recurrences: Backward Substitution

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

$T(n) = T(n/2) + 1$
$= [T(n/4) + 1] + 1$ /* expand T(n/2) */
$= T(n/4) + 2$ /* simplify */
$= [T(n/8) + 1] + 2$ /* expand T(n/4) */
$= T(n/8) + 3$ /* simplify */
$= \ldots$ . . .

$= T(n/2^{\lg n}) + \lg n$ /* $2^{\lg n} = n^{\lg 2} = n$ */
$= T(1) + \lg n$

$= c + \lg n$

$= O(\lg n)$

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.

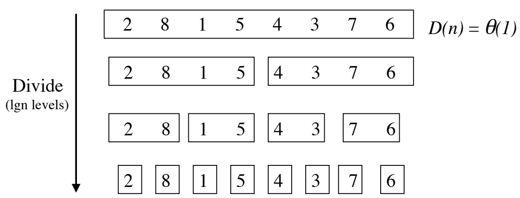
```
Merge-Sort(A,p,r)
if p < r then
   q ← ⌊(p+r)/2⌋
   Merge-Sort (A,p,q)
   Merge-Sort (A,q+1,r)
   Merge (A,p,q,r)

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

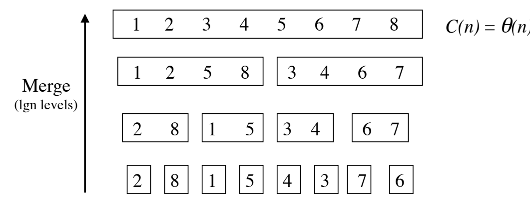
The Merge subroutine takes $\theta(n)$ time to merge n elements that are divided into two *sorted* arrays of $n/2$ elements each.

```
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



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

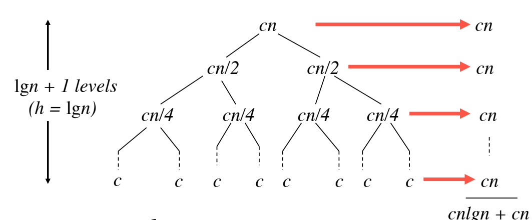
$$T(n) = \begin{cases} \theta(1) & \text{if } n = 1 \\ 2T(n/2) + \theta(n) & \text{otherwise} \end{cases}$$

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)

Recursion Tree for Merge-Sort



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

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

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

$$T(n) = \begin{cases} \theta(n^{\log_b a}) & \text{if } d < \log_b a \quad (a > b^d) \\ \theta(n^d \lg n) & \text{if } d = \log_b a \quad (a = b^d) \\ \theta(n^d) & \text{if } d > \log_b a \quad (a < b^d) \end{cases}$$

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

Alternate Version of Master Method

Master Theorem: 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 $T(n) = \theta(n^k)$

Solving Recurrences: Master Method

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

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

Solving Recurrences: Alt. Master Method

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

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

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

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

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