### 202 2016/17

1a Recursive squaring

Fibonacci numbers can be computed in logarithmic running time by using recursive squaring of a particular 2 ⨉ 2 matrix based on the following theorem:

{{Fn + 1, Fn}, {Fn, Fn - 1}} = {{1, 1}, {1, 0}}n

i) Use induction to prove the above theorem.

Assuming:

F0 = 0

F1 = 1

Fn = Fn - 1 + Fn - 2

Base case, n = 1 (since F-1 is undefined):

{{1, 1}, {1, 0}}1 = {{1, 1}, {1, 0}} ⇒

⇒ F2 = 1, F1 = 1, F0 = 0

Inductive step:

Inductive hypothesis (IH):

{{Fn + 1, Fn}, {Fn, Fn - 1}} = {{1, 1}, {1, 0}}n

To show:

{{Fn + 2, Fn + 1}, {Fn + 1, Fn}} = {{1, 1}, {1, 0}}n + 1

{{1, 1}, {1, 0}}n + 1 = {{1, 1}, {1, 0}}n ⨉ {{1, 1}, {1, 0}} =

(by IH) = {{Fn + 1, Fn}, {Fn, Fn - 1}} ⨉ {{1, 1}, {1, 0}} =

= {{Fn + 1 + Fn, Fn + 1}, {Fn + Fn - 1, Fn}} =

(by F) = {{Fn + 2, Fn + 1}, {Fn + 1, Fn}}

Hence the matrix exponentiation computes the Fibonacci series as required.

ii) Write down the recurrence equation for computing f(x, n) = xn that leads to a divide-and-conquer algorithm with logarithmic running time.

Implemented as:

if *n* is even

otherwise

Time complexity using master method:

⇒

b Substitution method

For the following two recurrences, guess the big-O upper bounds for their running times, and use the substitution method to prove the guesses are correct.

i)

Asymptotically equivalent to .

Guess: .

Inductive step:

IH:

(subs.)

(for c ≥ 1)

Base case: n = 2, c ≥ 2, T(2) = 4 ≤ 2(2log2)

ii)

Guess:

=> T(n) c - b

Inductive step:

IH:

(subs.)

c Master method

For the following four recurrences, use the master method to determine the running time complexity.

i)

ii)

iii)

iv)

d Recursion trees

Given the general form , recursion trees can be used to convert recurrences into trees whose nodes represent the computational costs at different levels.

i) How many leaf nodes does a recursion tree have?

n^logb(a)

ii) Write down the formula for summing up the operations of divide and combine over all recursion levels.

2a Activity-selection problem

The activity-selection problem is about scheduling several competing activities that require exclusive use of a common resource. The goal is to select a maximum-size set of mutually compatible activities.

i) Prove that a greedy strategy of selecting the activity of least duration is not guaranteed to produce an optimal solution.

Counter-example, with activities:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| A1 |  |  |  |  |  |  |
| A2 |  |  |  |  |  |  |
| A3 |  |  |  |  |  |  |

Here the optimal set is {A2, A3}. A greedy strategy choosing the least duration activity produces {A1}.

ii) Prove that a greedy strategy of selecting the activity with earliest start time is not guaranteed to produce an optimal solution.

Counter-example, with activities:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| A1 |  |  |  |  |  |  |
| A2 |  |  |  |  |  |  |
| A3 |  |  |  |  |  |  |

Here the optimal set is {A2, A3}. A greedy strategy choosing the earliest start time activity produces {A1}.

2b Apple picking

Given a table *A* with *m ⨉ n* cells, each having a certain quantity of apples. You start from the upper-left corner. At each step you can go down or right one cell.

i) Devise a bottom-up dynamic programming algorithm max\_apples(A) that computes the maximum number of apples you can collect.

Assuming *m* rows and *n* columns in notation.

We use a secondary table, *B*, with *m ⨉ n* cells as well. *Bi,j* (1 ≤ i ≤ m, 1 ≤ j ≤ n) will represent the highest number of apples we can collect in the sub table *Ai...m,j...n* (i.e. how many apples can we collect in *A* if we start at row *i* and column *j*), **excluding apples at *A*ij**.

max\_apples(A):

m := rows of A

n := columns of A

B := *m ⨉ n* table, filled with 0s

for i from m to 1

for j from n to 1

if i > 1

Bi-1,j := max(Bi-1,j, Ai,j + Bi,j)

if j > 1

Bi,j-1 := max(Bi,j-1, Ai,j + Bi,j)

return A1,1 + B1,1

ii) What is the running time complexity of max\_apples(A)?

The time complexity is O(mn). Aside from the obvious loops, initialising B is also O(mn), but this only affects the constant multiplier of the algorithm’s time complexity.

2c Radix search

Given the following keys together with their binary representations:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| A | Y | G | X | F |
| 00001 | 11001 | 00111 | 11000 | 00110 |

i) Draw the Binary Trie that results when you insert the above keys from left to right into an initially empty trie.

(In the graphs below, left child = 0 in key, right child = 1 in key)

Inserting A:

(A)

Inserting Y:

.

/ \

(A) (Y)

Inserting G:

.

/ \

. (Y)

/

.

/ \

(A) (G)

Inserting X:

.

/ \

. .

/ \

. .

/ \ /

(A) (G) .

/

.

/ \

(X) (Y)

Inserting F:

.

/ \

. .

/ \

. .

/ \ /

(A) . .

\ /

. .

/ \ / \

(F) (G) (X) (Y)

(Note: for BSTrie the order of insertion does not matter.)

ii) Draw the Patricia Trie that results when you insert the above keys from left to right into an initially empty trie.

Notation: (L,BC,R); L = left pointer, B = bit to test, C = current node, R = right pointer

Inserting A:

(-,4A,A)

Inserting Y:

(A,0Y,Y)

/

(-,4A,A)

Inserting G:

(A,0Y,Y)

/

(A,2G,G)

/

(-,4A,A)

Inserting X:

(A,0Y,X)

/ \

(A,2G,G) (X,4X,Y)

/

(-,4A,A)

Inserting F:

(A,0Y,X)

/ \

(A,2G,F) (X,4X,Y)

/ \

(-,4A,A) (F,4F,G)

2d Boyer-Moore string matching

Complete the last row of the good suffix rule table given below.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| i | 1 | 2 | 3 | 4 | 5 | 6 |
| P[i] | E | Y | P | E | Y | ϵ |
| gsr[i] | 3 | 3 | 3 | 5 | 1 | - |