Exercise Session 3

Data Structures and Algorithms, D-MATH, ETH Zurich

Program of today

Feedback of last exercise

Analyse the running time of (recursive) Functions

Solving Simple Recurrence Equations

Sorting Algorithms

1. Feedback of last exercise

2. Analyse the running time of (recursive) Functions

Analysis

How many calls to `f()`?

```
for(unsigned i = 1; i <= n/3; i += 3)
  for(unsigned j = 1; j <= i; ++j)
    f();
```

Analysis

How many calls to `f()`?

```
for(unsigned i = 1; i <= n/3; i += 3)
  for(unsigned j = 1; j <= i; ++j)
    f();
```

The code fragment implies $\Theta(n^2)$ calls to `f()`: the outer loop is executed $n/9$ times and the inner loop contains i calls to `f()`

How many calls to `f()`?

```
for(unsigned i = 0; i < n; ++i) {
  for(unsigned j = 100; j*j >= 1; --j)
    f();
  for(unsigned k = 1; k <= n; k *= 2)
    f();
}
```

How many calls to `f()`?

```
for(unsigned i = 0; i < n; ++i) {
  for(unsigned j = 100; j*j >= 1; --j)
    f();
  for(unsigned k = 1; k <= n; k *= 2)
    f();
}
```

We can ignore the first inner loop because it contains only a constant number of calls to `f()`

How many calls to `f()`?

```
for(unsigned i = 0; i < n; ++i) {
  for(unsigned j = 100; j*j >= 1; --j)
    f();
  for(unsigned k = 1; k <= n; k *= 2)
    f();
}
```

We can ignore the first inner loop because it contains only a constant number of calls to `f()`
The second inner loop contains $\lfloor \log_2(n) \rfloor + 1$ calls to `f()`. Summing up yields $\Theta(n \log(n))$ calls.

How many calls to `f()`?

```
void g(unsigned n) {
  if (n>0){
    g(n-1);
    f();
  }
}
```

How many calls to `f()`?

```
void g(unsigned n) {
  if (n>0){
    g(n-1);
    f();
  }
}
```

$$M(n) = M(n-1) + 1 = M(n-2) + 2 = \ldots = M(0) + n = n \in \Theta(n)$$

How many calls to `f()`?

```
// pre: n is a power of 2
//      n = 2^k
void g(int n){
  if (n>0){
    g(n/2);
    f()
  }
}
```

How many calls to `f()`?

```
// pre: n is a power of 2
//      n = 2^k
void g(int n){
  if (n>0){
    g(n/2);
    f()
  }
}
```

$$M(n) = 1 + M(n/2) = 1 + 1 + M(n/4) = k + M(n/2^k) \in \Theta(\log n)$$

How many calls to `f()`?

```
// pre: n is a power of 2
void g(int n){
  if (n>0){
    f();
    g(n/2);
    f();
    g(n/2);
  }
}
```

How many calls to `f()`?

```
// pre: n is a power of 2
void g(int n){
  if (n>0){
    f();
    g(n/2);
    f();
    g(n/2);
  }
}
```

$$M(n) = 2M\left(\frac{n}{2}\right) + 2 = 4M\left(\frac{n}{4}\right) + 4 + 2 = 8M\left(\frac{n}{8}\right) + 8 + 4$$
$$= n + n/2 + \ldots + 2 \in \Theta(n)$$

How many calls to f()?

```
// pre: n is a power of 2
//      n = 2^k
void g(int n){
  if (n>0){
    g(n/2);
    g(n/2);
  }
  for (int i = 0; i < n; ++i){
    f();
  }
}
```

How many calls to `f()`?

```
// pre: n is a power of 2
//      n = 2^k
void g(int n){
  if (n>0){
    g(n/2);
    g(n/2);
  }
  for (int i = 0; i < n; ++i){
    f();
  }
}
```

$$M(n) = 2M(n/2) + n = 4M(n/4) + n + 2n/2 = \ldots = (k+1)n \in \Theta(n \log n)$$

How many calls to f()?

```
void g(unsigned n) {
  for (unsigned i = 0; i<n ; ++i) {
    g(i)
  }
  f();
}
```

How many calls to `f()`?

```
void g(unsigned n) {
  for (unsigned i = 0; i<n ; ++i) {
    g(i)
  }
  f();
}
```

$T(0) = 1$

How many calls to `f()`?

```
void g(unsigned n) {
  for (unsigned i = 0; i<n ; ++i) {
    g(i)
  }
  f();
}
```

$T(0) = 1$
$T(n) = 1 + \sum_{i=0}^{n-1} T(i)$

How many calls to `f()`?

```
void g(unsigned n) {
  for (unsigned i = 0; i<n ; ++i) {
    g(i)
  }
  f();
}
```

$T(0) = 1$
$T(n) = 1 + \sum_{i=0}^{n-1} T(i)$

n	0	1	2	3	4
$T(n)$	1	2	4	8	16

How many calls to f()?

```
void g(unsigned n) {
  for (unsigned i = 0; i<n ; ++i) {
    g(i)
  }
  f();
}
```

$T(0) = 1$
$T(n) = 1 + \sum_{i=0}^{n-1} T(i)$

n	0	1	2	3	4
$T(n)$	1	2	4	8	16

Hypothesis: $T(n) = 2^n$.

Induction

Hypothesis: $T(n) = 2^n$.
Induction step:

$$T(n) = 1 + \sum_{i=0}^{n-1} 2^i$$
$$= 1 + 2^n - 1 = 2^n$$

How many calls to `f()`?

```
void g(unsigned n) {
  for (unsigned i = 0; i<n ; ++i) {
    g(i)
  }
  f();
}
```

You can also see it directly:

$$T(n) = 1 + \sum_{i=0}^{n-1} T(i)$$

$$\Rightarrow T(n-1) = 1 + \sum_{i=0}^{n-2} T(i)$$

$$\Rightarrow T(n) = T(n-1) + T(n-1) = 2T(n-1)$$

3. Solving Simple Recurrence Equations

Recurrence Equation

$$T(n) = \begin{cases} 2T(\frac{n}{2}) + \frac{n}{2} + 1, & n > 1 \\ 3 & n = 1 \end{cases}$$

Specify a closed (non-recursive), simple formula for $T(n)$ and prove it using mathematical induction. Assume that n is a power of 2.

Recurrence Equation

$$
\begin{aligned}
T(2^k) &= 2T(2^{k-1}) + 2^k/2 + 1 \\
&= 2(2(T(2^{k-2}) + 2^{k-1}/2 + 1) + 2^k/2 + 1 = \ldots \\
&= 2^k T(2^{k-k}) + \underbrace{2^k/2 + \ldots + 2^k/2}_{k} + 1 + 2 + \ldots + 2^{k-1} \\
&= 3n + \frac{n}{2}\log_2 n + n - 1
\end{aligned}
$$

$\Rightarrow$ Assumption $T(n) = 4n + \frac{n}{2}\log_2 n - 1$

Induction

1. Hypothesis $T(n) = f(n) := 4n + \frac{n}{2}\log_2 n - 1$
2. Base Case $T(1) = 3 = f(1) = 4 - 1.$
3. Step $T(n) = f(n) \longrightarrow T(2 \cdot n) = f(2n)$ ($n = 2^k$ for some $k \in \mathbb{N}$):

$$\begin{aligned}
T(2n) &= 2T(n) + n + 1 \\
&\overset{i.h.}{=} 2(4n + \frac{n}{2}\log_2 n - 1) + n + 1 \\
&= 8n + n\log_2 n - 2 + n + 1 \\
&= 8n + n\log_2 n + n\log_2 2 - 1 \\
&= 8n + n\log_2 2n - 1 \\
&= f(2n).
\end{aligned}$$

Master Method

$$T(n) = \begin{cases} aT(\frac{n}{b}) + f(n) & n > 1 \\ f(1) & n = 1 \end{cases} \qquad (a, b \in \mathbb{N}^+)$$

1. $f(n) = \mathcal{O}(n^{\log_b a - \epsilon})$ für eine Konstante $\epsilon > 0 \implies T(n) \in \Theta(n^{\log_b a})$
2. $f(n) = \Theta(n^{\log_b a}) \implies T(n) \in \Theta(n^{\log_b a} \log n)$
3. $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, und wenn $af(\frac{n}{b}) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large $n \implies T(n) \in \Theta(f(n))$

Examples

Maximum Subarray / Mergesort

$$T(n) = 2T(n/2) + \Theta(n)$$

Examples

Maximum Subarray / Mergesort

$$T(n) = 2T(n/2) + \Theta(n)$$

$$a = 2, b = 2, \ f(n) = cn = cn^1 = cn^{\log_2 2} \overset{[2]}{\Longrightarrow} T(n) = \Theta(n \log n)$$

Examples

Naive Matrix Multiplication Divide & Conquer[1]

$$T(n) = 8T(n/2) + \Theta(n^2)$$

[1]Treated in the course later on

Examples

Naive Matrix Multiplication Divide & Conquer[1]

$$T(n) = 8T(n/2) + \Theta(n^2)$$

$$a = 8, b = 2,\ f(n) = cn^2 \in \mathcal{O}(n^{\log_2 8 - 1}) \overset{[1]}{\Longrightarrow} T(n) \in \Theta(n^3)$$

[1] Treated in the course later on

Examples

Strassens Matrix Multiplication Divide & Conquer[2]

$$T(n) = 7T(n/2) + \Theta(n^2)$$

[2] Treated in the course later on

Examples

Strassens Matrix Multiplication Divide & Conquer[2]

$$T(n) = 7T(n/2) + \Theta(n^2)$$

$$a = 7, b = 2,\ f(n) = cn^2 \in \mathcal{O}(n^{\log_2 7-\epsilon}) \xRightarrow{[1]} T(n) \in \Theta(n^{\log_2 7}) \approx \Theta(n^{2.8})$$

[2]Treated in the course later on

Examples

$$T(n) = 2T(n/4) + \Theta(n)$$

Examples

$$T(n) = 2T(n/4) + \Theta(n)$$

$$a = 2, b = 4,\ f(n) = cn \in \Omega(n^{\log_4 2 + 0.5}),\ 2f(n/4) = c\frac{n}{2} \leq \frac{c}{2}n^1 \overset{[3]}{\Longrightarrow} T(n) \in \Theta(n)$$

Examples

$$T(n) = 2T(n/4) + \Theta(n^2)$$

Examples

$$T(n) = 2T(n/4) + \Theta(n^2)$$

$a = 2, b = 4$, $f(n) = cn^2 \in \Omega(n^{\log_4 2 + 1.5})$, $2f(n/4) = \frac{n^2}{8} \leq \frac{1}{8}n^2 \overset{[3]}{\Longrightarrow}$
$T(n) \in \Theta(n^2)$

4. Sorting Algorithms

Quiz

Consider the following three sequences of snap-shots (steps) of the algorithms (a) Insertion Sort, (b) Selection Sort and (c) Bubblesort. Below each sequence provide the corresponding algorithm name.

5	4	1	3	2
1	4	5	3	2
1	2	5	3	4
1	2	3	5	4
1	2	3	4	5

5	4	1	3	2
4	1	3	2	5
1	3	2	4	5
1	2	3	4	5

5	4	1	3	2
4	5	1	3	2
1	4	5	3	2
1	3	4	5	2
1	2	3	4	5

Quiz

Consider the following three sequences of snap-shots (steps) of the algorithms (a) Insertion Sort, (b) Selection Sort and (c) Bubblesort. Below each sequence provide the corresponding algorithm name.

5	4	1	3	2
1	4	5	3	2
1	2	5	3	4
1	2	3	5	4
1	2	3	4	5

5	4	1	3	2
4	1	3	2	5
1	3	2	4	5
1	2	3	4	5

5	4	1	3	2
4	5	1	3	2
1	4	5	3	2
1	3	4	5	2
1	2	3	4	5

Quiz

Consider the following three sequences of snap-shots (steps) of the algorithms (a) Insertion Sort, (b) Selection Sort and (c) Bubblesort. Below each sequence provide the corresponding algorithm name.

5	4	1	3	2
1	4	5	3	2
1	2	5	3	4
1	2	3	5	4
1	2	3	4	5

selection

5	4	1	3	2
4	1	3	2	5
1	3	2	4	5
1	2	3	4	5

bubblesort

5	4	1	3	2
4	5	1	3	2
1	4	5	3	2
1	3	4	5	2
1	2	3	4	5

insertion

Quiz

Execute two further iterations of the algorithm Quicksort on the following array.
The first element of the (sub-)array serves as the pivot.

8	7	10	15	3	6	9	5	2	13
2	7	5	6	3	**8**	9	15	10	13

Quiz

Execute two further iterations of the algorithm Quicksort on the following array.
The first element of the (sub-)array serves as the pivot.

8	7	10	15	3	6	9	5	2	13
2	7	5	6	3	**8**	9	15	10	13

Quiz

Execute two further iterations of the algorithm Quicksort on the following array.
The first element of the (sub-)array serves as the pivot.

8	7	10	15	3	6	9	5	2	13
2	7	5	6	3	**8**	9	15	10	13
2	7	5	6	3	**8**	**9**	15	10	13
2	3	5	6	**7**	**8**	**9**	13	10	**15**

Algorithm NaturalMergesort(A)

Input: Array A with length $n > 0$
Output: Array A sorted

```
repeat
    r ← 0
    while r < n do
        l ← r + 1
        m ← l; while m < n and A[m+1] ≥ A[m] do m ← m + 1
        if m < n then
            r ← m + 1; while r < n and A[r+1] ≥ A[r] do r ← r + 1
            Merge(A, l, m, r);
        else
            r ← n
until l = 1
```

Quicksort with logarithmic memory consumption

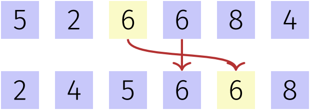
Input: Array A with length n. $1 \leq l \leq r \leq n$.
Output: Array A, sorted between l and r.

while $l < r$ **do**
    Choose pivot $p \in A[l, \ldots, r]$
    $k \leftarrow$ Partition$(A[l, \ldots, r], p)$
    if $k - l < r - k$ **then**
        Quicksort$(A[l, \ldots, k-1])$
        $l \leftarrow k + 1$
    else
        Quicksort$(A[k+1, \ldots, r])$
        $r \leftarrow k - 1$

The call of Quicksort$(A[l, \ldots, r])$ in the original algorithm has moved to iteration (tail recursion!): the if-statement became a while-statement.

Stable and in-situ sorting algorithms

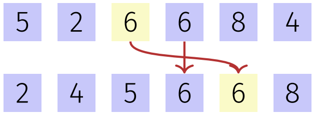
- Stable sorting algorithms don’t change the relative position of two equal elements.



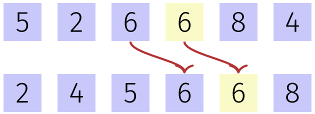
not stable

Stable and in-situ sorting algorithms

- Stable sorting algorithms don't change the relative position of two equal elements.



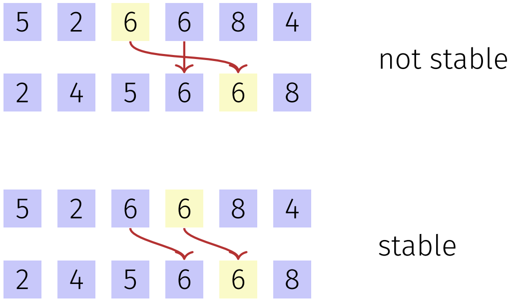
not stable



stable

Stable and in-situ sorting algorithms

- Stable sorting algorithms don't change the relative position of two equal elements.



- In-situ algorithms require only a constant amount of additional memory.

Which of the sorting algorithms are stable? Which are in-situ? (How) can we make them stable / in-situ?

Questions?