Tutorial 2
Data Structures ’08

Jonathan Cederberg <jonathan.cederberg@it.uu.se>

Monday, September 21th, 2008
Outline

1. First assignment
2. Probability
3. Sorting
4. Invariants
5. Second assignment
Outline

1. First assignment
2. Probability
3. Sorting
4. Invariants
5. Second assignment
Review of assignment 1

\[
\begin{align*}
&n \quad 4^{\lg n} \quad n! \quad \lg n \quad 2^{2^n} \quad (n + 1)! \quad \left(\frac{3}{2}\right)^n \\
n^3 \quad n \lg n \quad 2^n \quad (\lg n)^{\lg n} \quad n \cdot 2^n
\end{align*}
\]
Review of assignment 1

\[ n \quad 4^{\log n} \quad n! \quad \log n \quad 2^n \quad (n + 1)! \quad \left(\frac{3}{2}\right)^n \]

\[ n^3 \quad n^{\log n} \quad 2^n \quad (\log n)^{\log n} \quad n \cdot 2^n \]

1. Do not compare the output of functions!
Review of assignment 1

\[ n \quad 4^{|\lg n|} \quad n! \quad |\lg n| \quad 2^{2^n} \quad (n + 1)! \quad \left(\frac{3}{2}\right)^n \]

\[ n^3 \quad n \lg n \quad 2^n \quad (\lg n)^{|\lg n|} \quad n \cdot 2^n \]

1. Do not compare the output of functions!
2. Do not use calculus! The functions are not defined on \( \mathbb{R} \)!
Review of assignment 1

\[ n \ 4^{\log n} \ 4 \ log n \ 2^{2n} \ (n + 1)! \ \left(\frac{3}{2}\right)^n \]

\[ n^3 \ n \log n \ 2^n \ (\log n)^{\log n} \ n \cdot 2^n \]

1. Do not compare the output of functions!
2. Do not use calculus! The functions are not defined on \( \mathbb{R} \)!
3. Use the definition of \( O \)!
Outline

1. First assignment
2. Probability
3. Sorting
4. Invariants
5. Second assignment
Review of probability theory

- A *sample space* $S$ is the set of all possible outcomes.
- An *event* $A$ is a subset $A \subseteq S$
- A random variable is often denoted $X$
- The *expected value* $E[X]$ of a random variable $X$ is defined as
  \[ E[X] = \sum_x x \cdot \Pr\{X = x\} \]
- Note that $E[\cdot]$ is a linear operator, i.e.
  \[ E \left[ \sum_{i=1}^{n} (a_i \cdot X_i) \right] = \sum_{i=1}^{n} (a_i \cdot E[X_i]) \]
Exercise: Bleaching

- You have a function, Biased-Random, that returns 1 with probability $p$ and 0 with probability $1 - p$. Sadly you do not know $p$. Design a function Unbiased-Random that returns 1 with probability $1/2$ and 0 with probability $1/2$. 

```plaintext
Unbiased-Random:

while true do
  x ← Biased-Random
  y ← Biased-Random
  if x ≠ y then return x
```

Jonathan Cederberg | jonathan.cederberg@it.uu.se
Exercise: Bleaching

- You have a function, Biased-Random, that returns 1 with probability $p$ and 0 with probability $1 - p$. Sadly you do not know $p$. Design a function Unbiased-Random that returns 1 with probability 1/2 and 0 with probability 1/2.

Unbiased-Random

1. while true
2. do
3. $x \leftarrow$ Biased-Random
4. $y \leftarrow$ Biased-Random
5. if $x \neq y$
6. then return $x$
Exercise: Bleaching (cont.)

- Why does this work?
- Because Unbiased-Random only returns when $x = 0$ and $y = 1$ or vice versa. Since
  \[
  \Pr\{x = 0 \land y = 1\} = (1 - p)p \\
  = p(1 - p) = \Pr\{x = 1 \land y = 0\}
  \]
  and there are no other outcomes, Unbiased-Random is fair.
- Note that this relies on that the calls to Biased-Random are independent.
Definition

- An *indicator random variable* \( I\{A\} \) of an event \( A \) is defined as

\[
I\{A\} = \begin{cases} 
1 & \text{if } A \text{ occurs} \\
0 & \text{if } A \text{ does not occur}
\end{cases}
\]

- By the definition of expected value we have for any indicator random variable \( X_A \)

\[
E[X_A] = E[I\{A\}] = 1 \cdot \Pr\{A\} + 0 \cdot \Pr\{S \setminus A\} = \Pr\{A\}
\]
Example: Flipping a coin

- $S = \{H, T\}$
- $A = \{H\}$
- $X_H = I\{\text{Flip is } H\} = \begin{cases} 1 & \text{if flip is } H \\ 0 & \text{if flip is } H \end{cases}$

$$E[X_H] = E[I\{\text{Flip is } H\}] = 1 \cdot \Pr\{\text{Flip is } H\} + 0 \cdot \Pr\{\text{Flip is not } H\} = 1 \cdot (1/2) + 0 \cdot (1/2) = 1/2$$
The Hiring Problem: You have $n$ candidates to the assistant job. You want to always keep the best person for the job.
The Hiring Problem: You have $n$ candidates to the assistant job. You want to always keep the best person for the job.

Hire-Assistant($n$)

1. $best \leftarrow 0$
2. for $i \leftarrow 1$ to $n$
3. do interview candidate $i$
4. if candidate $i$ is better than candidate $best$
5. then $best \leftarrow i$
6. hire candidate $i$
To evaluate the expected number of candidates that get hired, use indicator random variables:

- \( X_i = I\{\text{candidate } i \text{ is hired}\} = \begin{cases} 1 & \text{if candidate } i \text{ is hired} \\ 0 & \text{if candidate } i \text{ is not hired} \end{cases} \)

- \( X = [\text{the number of candidates hired}] = \sum_{i=1}^{n} X_i \)
- \( E[X_i] = ? \)
\[ E[X_i] = \frac{1}{i} \]

\[
E[X] = E\left[\sum_{i=1}^{n} X_i\right]
\]

\[
= \sum_{i=1}^{n} E[X_i]
\]

\[
= \sum_{i=1}^{n} \left(\frac{1}{i}\right)
\]

\[
= \log n + \mathcal{O}(1)
\]
Examples:

- What is the probability of hiring exactly once?
Examples:

- What is the probability of hiring exactly once?
- The first person is always hired. Therefore, the answer is 
  \[ \Pr\{\text{The first person is the best}\} = \frac{1}{n} \]
Examples (cont):

- What is the probability of hiring all $n$ persons?
Examples (cont):

- What is the probability of hiring all $n$ persons?
- The persons must come in reverse order, competencewise, giving

$$
\Pr\{\text{Hire all } n \text{ persons}\} = \Pr\{\text{Reversely ordered competencewise}\} \\
= \Pr\{1\text{st person worst}\} \cdot \ldots \cdot \Pr\{n\text{th person best}\} \\
= \prod_{i=1}^{n} \left(1/i\right) \\
= \frac{1}{n!}
$$
Outline

1. First assignment
2. Probability
3. Sorting
4. Invariants
5. Second assignment
Heapsort

- We know the “bubbling” behaviour of Max-Heapify is used for maintaining the heap property in $\Theta(lg n)$ in the worst case.
Heapsort

- We know the “bubbling” behaviour of Max-Heapify is used for maintaining the heap property in $\Theta(lg\ n)$ in the worst case.

- We know that Build-Max-Heap produces a max-heap by repeated calls to Max-Heapify, and that is $\Theta(n)$ in the worst case.
Heapsort

- We know the “bubbling” behaviour of Max-Heapify is used for maintaining the heap property in $\Theta(lg\ n)$ in the worst case.

- We know that Build-Max-Heap produces a max-heap by repeated calls to Max-Heapify, and that is $\Theta(n)$ in the worst case.

- We know the Heapsort algorithm basics: create a heap, take care of the biggest (smallest) element, Max-Heapify the rest of the elements.
Heapsort

- We know the “bubbling” behaviour of Max-Heapify is used for maintaining the heap property in $\Theta(\lg n)$ in the worst case.
- We know that Build-Max-Heap produces a max-heap by repeated calls to Max-Heapify, and that is $\Theta(n)$ in the worst case.
- We know the Heapsort algorithm basics: create a heap, take care of the biggest (smallest) element, Max-Heapify the rest of the elements.
- We know Heapsort is $O(n \lg n)$. 
**Heapsort** Pseudocode:

```plaintext
Heapsort(A)
1. Build-Max-Heap(A)
2. for i ← length[A] downto 2
4. heap-size[A] ← heap-size[A] − 1
5. Max-Heapify(A, 1)
```
Why is Heapsort $\Theta(n \lg n)$ in the worst case?

- Line 1 takes $\Theta(n)$.
- Line 2 to 5 is basically $n-1$ calls to Max-Heapify.
- The problem size decreases by 1 for each call to Max-Heapify, so it takes $c \cdot \sum_{i=2}^{n} \lg(i)$ time.
- This gives overall time of $\Theta(n) + c \cdot \sum_{i=2}^{n} \lg(i)$.
Lemma:
\[ \lg(n!) = \Theta(n \lg(n)) \]
Proof:
Exercise (Hint: use Stirling's approximation)
\[ c \cdot \sum_{i=2}^{n} \lg(i) = c \lg(\prod_{i=2}^{n} i) \]
\[ = c \cdot \lg(n!) \]
\[ = \Theta(n \lg(n)) \]
This gives that Heapsort is $\Theta(n) + \Theta(n \lg(n)) = \Theta(n \lg(n))$, and in particular, we have established the lower bound to be $\Omega(n \lg(n))$. 
Outline

1. First assignment
2. Probability
3. Sorting
4. Invariants
5. Second assignment
Invariant of Heapsort

**Heapsort** Pseudocode:

Heapsort(A)
1. Build-Max-Heap(A)
2. for i ← length[A] downto 2
4. heap-size[A] ← heap-size[A] − 1
5. Max-Heapify(A, 1)
**Invariant of Heapsort**

**Heapsort** Pseudocode:

Heapsort($A$)
1. Build-Max-Heap($A$)
2. for $i \leftarrow \text{length}[A]$ downto 2
4. heap-size[$A$] $\leftarrow$ heap-size[$A$] $- 1$
5. Max-Heapify($A$, 1)

**Invariant:** The array $A[i..\text{length}[A]]$ is always sorted in increasing order.
Outline

1. First assignment
2. Probability
3. Sorting
4. Invariants
5. Second assignment
Assignment 2 Implement Heapsort, the version based on Max-Heapify.