In This Course...

You are expected to learn:
- How to evaluate algorithms, and
- How to choose the most appropriate ones.

I will talk about:
- Analysis methods for algorithms, and
- Several representative and/or characteristic algorithms.

This Course will be Given in English

- **Pros**
  - You may understand English better than Japanese
  - You may have found yet another opportunity to learn English
- **Cons**
  - You may miss some of the information, because
    - Your English ability is limited, or
    - My English ability is limited
  - You may lose an opportunity to learn Japanese
  - You won't be able to hear my excellent jokes

Materials

- You can find copies of the slides shown in this class in my Web page.
  - http://www.logos.t.u-tokyo.ac.jp/~chik
  (in PDF; sorry, no animations)
- The topic areas of this course is probably too wide for a few textbooks, and is also too sparse to justify reading many books.
- You are encouraged to attend the classes, which is less costly and time-saving than reading many thick books, I hope.
Evaluation

- We do not expect you to understand every detail of every algorithms, but we'd like you to understand algorithms of at least one field in depth and, at the same time, become familiar with the general principles in designing and/or selecting algorithms.
- Evaluation will be based on your term project report to be submitted at the end of the term.
- The subject of the project is to apply at least one of the algorithms taken up in the class to a concrete problem, and compare its performance with other (probably more naive) algorithms.

Schedule

- Oct. 7th: Introduction & Complexity Theory
- Oct. 15th: Search Trees
- Oct. 21st: Combinatorial Optimization
- Oct. 28th: Heuristic Search
- Nov. 5th: Text Search
- Nov. 11th: Data Compression
- Nov. 18th: Memory Management
- Nov. 25th: Graph Algorithms
- Dec. 2nd: Graph Algorithms (cont.)
- Dec. 9th: Computational Geometry
- Dec. 16th: Concurrency Control (Jan. 15th: Expected to be Canceled)
- Jan. 20th: Clustering

Algorithm Analyses

- Correctness
- Performance: Computational Complexity
  - How much computational resource is required to obtain the required output for given input?
  - Complexity as a function of input size

Which Kinds of Resource?

- Time complexity
  - How much computation time is required
- Space/memory complexity
  - How much memory space is required
- Communication (only in parallel algorithms)
  - How much communication is required
  - The amount, frequency, sensitivity to delay

  Time complexity will be discussed mainly
Basic Assumptions

- The memory is flat
  - The same access time for all the main memory = random access
- Operations have similar costs
  - Costs of all the basic operations are the same → # of operations instead of time
These assumptions do not rigorously hold
- Memory hierarchy: caches, virtual memory
- Pipelining and/or super-scalar make even the same instruction take different time

Asymptotic Complexity

- Larger problems demand for larger computation
  ⇒ Complexity as a function of the problem size
  - $T(n)$ where $n$ indicates the amount and/or complexity of the input data
- Usually, the function $T(n)$ is monotonic
  - Computation is not costly for small problems, anyway
  ⇒ Large problems are our resource concern
  ⇒ What’s important is how $T(n)$ increases with $n$
  e.g. “with large $n$, the complexity is asymptotically proportional to $n \log n$”

Big-$O$ notation

- Means an upper bound of asymptotic complexity
  i.e., “computation is never larger than this”
- An algorithm is said to have the complexity of $O(f(n))$ when $\exists c \ni n_0 \forall n \geq n_0 T(n) \leq c f(n)$
  The actual complexity $T(n)$ for problem size $n$
  never exceeds $f(n)$ multiplied by a constant $c$
  for any $n$ greater than or equal to $n_0$
- Smaller $O$ means more precise analysis

Ω-notation

- Means a lower bound of asymptotic complexity
  i.e., “never less than this”
- An algorithm is said to have the complexity of $\Omega(f(n))$ when $\exists c \ni n_0 \forall n \geq n_0 T(n) \geq c f(n)$
  The actual complexity of $T(n)$ for problem size $n$
  never underrun $f(n)$ multiplied by a constant $c$
  for any $n$ greater than or equal to $n_0$
- Greater $\Omega$ means more precise analysis
**Θ-notation**

- An algorithm is said to have the complexity $\Theta(f(n))$ when both $O(f(n))$ and $\Omega(f(n))$ hold:
  - $\exists c \exists n_0 \forall n \geq n_0 \; T(n) \leq cf(n)$, and
  - $\exists c \exists n_0 \forall n \geq n_0 \; T(n) \geq cf(n)$

Note: Upper and lower bounds can be satisfied separately. As existential quantifiers are independent, two $c$'s and two $n_0$'s can be different.

- Upper and lower bounds agree except for a constant factor.

**Complexity Indices for More Complicated Cases**

- Complexity may depend on qualitative characteristics of input data.
  - Different indices for different purposes.

- Worst case complexity:
  - Computational resource enough for all cases.
  - E.g., mission-critical systems.

- Average case complexity:
  - Average of varying expected inputs.
  - Assumption on distribution of the input required.

- Amortized complexity (discussed later).

**Average vs. Worst Case Complexities**

- Binary search tree without balancing:
  - The worst case can be the same as a linear list.
  - A single search may require $O(n)$ time.
  - In most cases, imbalance is not so severe.
  - Average time complexity for uniformly random trees is $O(\log n)$.

**Multiple-Word Counter**

- A counter that can count up to a very big number.
  - The number does not fit in one memory word.
  - An array of integer words can be used.
  - When one word can express 0 to $m-1$, an integer $x$ can be expressed as
    $$x = \sum_{k=0}^{n-1} a[k] \times m^k$$
  - For simplicity, we assume that $m = 2$ (1 bit/word).
Multiple Word Counter: Counting Up

A procedure for counting up would be as follows.

```java
increment(int a[]) {
  int k;
  for (k=0; a[k] == 1; k++)
    a[k] = 0;
  a[k] = 1;
}
```

Complexity of Counting Up

- **Worst Case Complexity**
  - When the value is $2^k - 1$, carry propagation of $k$ bits will take place, which may take long
    \[ \Rightarrow O(k), \text{i.e. } O(\log n) \]
  - This, however, rarely happens for large $k$
- **Average complexity:** Counting up to $n = 2^k - 1$
  - $a[0]$ always changes $2^k$
  - $a[1]$ changes every other times $2^{k-1}$
  - $a[j]$ changes every $2^j$ times $2^{k-j}$
    \[ \Rightarrow \text{Twice for each step in average } \Rightarrow O(1) \]

Shortcomings of Average Complexity

- The average complexity of counting up to $n$ is $O(n)$, thus the complexity for each increment is constant, in average.
- This, however, does not guarantee anything on the complexity of individual increments.
- Even when the first several increments has the complexity of $O(n)$, the total remains to be $O(n)$ if all others have the complexity of $O(1)$.
- This is not useful when the total number of increments is not known.

What Mandates Carry Propagation?

- Many bits of carry propagation is required sometimes, but such cases are rare.
- Before an increment requiring many bits of carry propagation, many increments without many bits of carry propagation should precede.
- Increments without many bits of carry propagation are the cause of many carries.
  \[ \Rightarrow \text{These operation should pay the cost!} \]
**Amortized Complexity**

For algorithms with sequences of operations, most of which are efficient but some are quite costly.
- The upper bound $T(n)$ of a sequence of first $n$ operations is analyzed.
- The average cost of the first $n$ operations is $T(n)/n$, which can be considered to be the cost of a single operation during this period.
- Some inexpensive operations may actually have much less costs than $T(n)/n$, but all operations are charged the same amount.

**What is Amortization?**

The gradual elimination of a liability, such as a mortgage, in regular payments over a specified period of time. Such payments must be sufficient to cover both principal and interest.

http://www.investorwords.com/

- In amortized complexity, the cost is regularly paid, for both low-cost and high-cost operations.
- Unlike amortized loan, which is gradual payment of debt, amortized algorithms reserves the fund for future costs in advance.

**Amortized Analysis for the Multi-Word Counter**

Observations
- Bit flipping costs proportional to the # of bits.
- Bits of one are turned to zero sometime in future.
- A single increment always turns one zero to one.

Principles
- The cost of an increment is estimated to be two units: an actual inversion + a future inversion.
- When no carry propagation is required, the actual cost is one unit, that of inverting a single bit.
- The remaining cost of one unit is reserved.
- Carry propagation costs are paid from this reserve.
### Amortization of Carry Propagation Costs

<table>
<thead>
<tr>
<th>Bits</th>
<th>Cost</th>
<th>Cost sum</th>
<th>Paid sum</th>
<th>Reserved</th>
</tr>
</thead>
<tbody>
<tr>
<td>0→1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>→2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>→3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>→4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>→5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>→6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>→7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>→8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### More Realistic Example: Adjusting Array Sizes

- Arrays have random-access, which is good, but requires a contiguous memory area.
- When the required number of elements cannot be predicted beforehand, we may want to dynamically adjust the size.
- Adjacent memory area may be already in use: Allocating a new array and copying all the elements are necessary.
  Doesn't this sound too costly?

### Expandable Array: Constant Amount Expansion

```c
int size = initial size, used = 0;
entry *table = An array of the initial size;
void add_to_table(entry x) {
    if (used == size) {
        size += some constant;
        table = a new array of size size;
        Copy all the elements to the new array;
    }
    table[used++] = x;
}
```

### Costs of Expansion

- The array is initially made small, repeatedly expanded by \( c \) elements to \( m \) elements finally. The number of expansions is \( m/c \) times, which is proportional to the final size \( m \).
- The expansions cost proportional to the number of elements, which linearly approaches \( m \).
- Thus the total cost is proportional to \( m^2 \).
Expandable Array: Constant Ratio Expansion

```c
int size = initial size, used = 0;
entry *table = An array of the initial size;
void add_to_table(entry x) {
    if (used == size) {
        size *= some constant greater than 1;
        table = a new array of size size;
        Copy all the elements to the new array;
    }
    table[used++] = x;
}
```

Actual Computation Costs

- Assignment to an array element: \( r \)
- Allocation cost per element: \( a \)
- Copying cost per element: \( c \)

Expanding an array of size \( 2^k \) to \( 2^{k+1} \) requires:

- \( 2^{k+1}a \) for allocation of a new array, and
- \( 2^kc \) for copying the elements.

Thus, the total cost of adding an element is:

- Usually \( r \), but,
- \( 2^{k+1}a + 2^kc \) of extra when the size is \( 2^k \)

Amortized Analysis

- After expanding the array to have \( 2^k \) elements,
  \( 2^{k-1} \) more elements can be added to the array
  before the next expansion.
- The extra cost of the next expansion is \( 2^{k-1}a + 2^c \).
- This cost should be reserved before the next expansion, that is, while \( 2^{k-1} \) elements are added to the array.

```
<table>
<thead>
<tr>
<th>( 2^{k-1} )</th>
<th>( 2^{k-2} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( 2^{k-1} )</td>
<td>( 2^{k-2} )</td>
</tr>
<tr>
<td>( 2^k )</td>
<td>( 2^k )</td>
</tr>
</tbody>
</table>
```

Amortized Analysis (cont.)

- During \( 2^{k-1} \) element additions, \( 2^{k-1}a + 2^c \) should be reserved for the next expansion.
- Thus, each of the addition should reserve \( (2^{k-1}a + 2^c) / 2^{k-1} = 4a + 2c \).
- Actual assignment cost without the reserve is \( r \).
- In total, adding an element should pay the cost:
  \( 4a + 2c + r \),
  which is a constant.

Expandable arrays can be implemented with amortized complexity of \( O(1) \) per operation.
Comparing Const. Amount & Const. Ratio
- Time complexities for adding $m$ elements
  - Const. amount: $O(m^2)$
  - Const. ratio: $O(m)$
- Space complexities for $m$ elements
  - Both are $O(m)$, ignoring constant factors.

Shrinkable Arrays
- When the number of elements decreases, we may want to make the array smaller.
- When half is used, should we halve the array size?
  ⇒ Repeating increment and decrement between half and half+1 size is quite costly.
- When the usage decreased to a quarter, we can halve the array size half.
  ⇒ This gives amortized complexity of $O(1)$
- In general, introducing hysteresis may be useful.

Expandable/Shrinkable Array

Shortcomings of Amortized Analysis
- Limited information on costs of individual operations
- When all we know is the amortized complexity of an algorithm to be $O(f(n))$ for each of operations, we have to expect, in the worst case, that the next operation may cost $\sum_{k=1}^{n} f(k)$.
- No guarantee of real-time responses.
See You Next Tuesday

- October 14th is a national holiday (Health and Sports Day). No classes given on this day; Enjoy sports and improve your health!
- October 15th is a Tuesday, but classes originally scheduled on Mondays, including this class, will be given on this day.
- The same applies again to November 5th.

Materials

- You can find copies of the slides used in this class in the following Web page.
  http://www.logos.t.u-tokyo.ac.jp/~chik
  (in PDF; sorry, no animations)