First Chapter

- In this chapter, we will answer the following two questions
  - What does it mean to be an efficient algorithm?
  - How can one tell that it is more efficient than other algorithms?

  based on some easy-to-understand searching and sorting algorithms that we may have seen earlier.

Searching Problem

- Assume A is an array with n elements A[1], A[2], … A[n]. For a given element x, we must determine whether there is an index *j*; $1 \le j \le n$, such that $x = A[j]$
- Two algorithms, among others, address this problem
  - Linear Search
  - Binary Search

Linear Search Algorithm

Algorithm: LINEARSEARCH

Input: array $A[1..n]$ of n elements and an element x.

Output: j if $x = A[j]$, $1 \le j \le n$, and 0 otherwise.

```
1. j ← 1
2. while (j < n) and (x ≠A[j])
3.    j ← j + 1
4. end while
5. if x = A[j] then return j else return 0
```


Analyzing Linear Search

- One way to measure efficiency is to count how many statements get executed before the algorithm terminates
- One should keep an eye, though, on statements that are executed "repeatedly".
- What will be the number of "element" comparisons if
  - X first appears in the first element of A
  - X first appears in the middle element of A
  - X first appears in the last element of A
  - X doesn't appear in A.

Binary Search

- We can do "better" than linear search if we knew that the elements of A are sorted, say in non-decreasing order.
- The idea is that you can compare x to the middle element of A, say A[middle].
  - If x < A[middle] then you know that x cannot be an element from A[middle+1], A[middle+2], …A[n]. Why?
  - If x > A[middle] then you know that x cannot be an element from A[1], A[2], …A[middle-1]. Why?

Binary Search Algorithm

Algorithm: BINARYSEARCH

Input: An array $A[1..n]$ of n elements sorted in nondecreasing order and an element x.

Output: j if $x = A[j]$, $1 \le j \le n$, and 0 otherwise.

```
1. low ← 1; high ← n; j ← 0
2. while (low ≤ high) and (j = 0)
3.    mid ← ⌊(low + high)/2⌋
4.    if x = A[mid] then j ← mid
5.    else if x < A[mid] then high ← mid - 1
6.    else low ←  mid + 1
7. end while
8. return j
```


Worst Case Analysis of Binary Search

- What to do: Find the maximum number of element comparisons
- How to do:
  - The number of "element" comparisons is equal to the number of iterations of the while loop in steps 2-7. HOW?
  - How many elements of the input do we have in the
    - First iteration
    - Second iteration
    - Thrid iteration
    - …
    - i^{th} iteration
  - The last iteration occurs when the size of input we have =

Theorem

- The number of comparisons performed by Algorithm BINARYSEARCH on a sorted array of size n is at most $\lfloor \log n \rfloor + 1$

Merging Two Sorted Lists

- Problem Description: Given two lists (arrays) that are sorted in non-decreasing order, we need to ***merge*** them into one list sorted in non-decreasing order.
- Example:

Input

3	7	9	12

1	2	4	13	14

Output

1	2	3	4	7	9	12	13	14

How to merge two arrays?

B[0..5]	A[0..2]	A[3..5]
	2 4 5	3 7 8
2	2 4 5	3 7 8
2 3	2 4 5	3 7 8
2 3 4	2 4 5	3 7 8
2 3 4 5	2 4 5	3 7 8
2 3 4 5 7 8	2 4 5	3 7 8

Algorithm MERGE

Algorithm: MERGE

Input: An array $A[1..m]$ of elements and three indices p, q and r, with $1 \le p \le q < r \le m$, such that both the subarrays $A[p..q]$ and $A[q+1..r]$ are sorted individually in nondecreasing order.

Output: $A[p..r]$ contains the result of merging the two subarrays $A[p..q]$ and $A[q+1..r]$.

Comment: $B[p..r]$ is an auxiliary array.

Algorithm MERGE (Cont.)

1. $s \leftarrow p$; $t \leftarrow q + 1$; $k \leftarrow p$
2. while $s \le q$ and $t \le r$
3. if $A[s] \le A[t]$ then
4. $B[k] \leftarrow A[s]$
5. $s \leftarrow s + 1$
6. else
7. $B[k] \leftarrow A[t]$
8. $t \leftarrow t + 1$
9. end if
10. $k \leftarrow k + 1$
11. end while
12. if $(s = q + 1)$ then $B[k..r] \leftarrow A[t..r]$
13. else $B[k..r] \leftarrow A[s..q]$
14. end if
15. $A[p..r] \leftarrow B[p..r]$

Analyzing MERGE

- Assuming arrays A[p,q] and A[q+1,r]
  - The least number of comparisons is which occurs when
  - The most number of comparisons is which occurs when
  - The number of element assignments performed is

Selection Sort

Algorithm: *SELECTIONSORT*
Input: An array $A[1..n]$ of n elements.
Output: $A[1..n]$ sorted in nondecreasing order.

```
1. for i ← 1 to n - 1
2.   k ← i
3.   for j ← i + 1 to n
           {Find the index of the i^th smallest element}
4.      if A[j] < A[k] then k ← j
5.   end for
6.   if k ≠ i then interchange A[i] and A[k]
7. end for
```


Selection Sort Example

i	k					
		5	2	9	8	4
1	2	2	5	9	8	4
2	5	2	4	9	8	5
3	5	2	4	5	8	9
4	4	2	4	5	8	9

Analyzing Selection Sort

- We need to find the number of comparisons carried out in line #4:
  - For each iteration of the outer for loop, how many times is line #4 executed?
  - Therefore, in total, line #4 is executed
- The number of element Interchanges (swaps):
  - Minimum:
  - Maximum:
  - ❖ NOTE: The number of element assignments is 3 times the number of element interchanges

Insertion Sort

Algorithm: ***INSERTIONSORT***
Input: An array $A[1..n]$ of n elements.
Output: $A[1..n]$ sorted in nondecreasing order.

```
1. for i ← 2 to n
2.   x ← A[i]
3.   j ← i - 1
4.   while (j > 0) and (A[j] > x)
5.     A[j + 1] ← A[j]
6.     j ← j - 1
7.   end while
8.   A[j + 1] ← x
9. end for
```


Insertion Sort Example

x=2	5	2	9	8	4
x=9	2	5	9	8	4
x=8	2	5	9	8	4
x=4	2	5	8	9	4
	2	4	5	8	9

Analyzing Insertion Sort

- The minimum number of element comparisons is which occurs when
- The maximum number of element comparisons is which occurs when
- The number of element assignments is

Bottom-Up Merge Sort

- Informally, the algorithm does the following
  - 1. Divide the array into pairs of elements (with possibly single elements in case the number of elements is)
  - 2. Merge each pair in non-decreasing order (with possibly a single “pair” left)
  - 3. Repeat step 2 until there is only one “pair” left.

Bottom-Up Merge Sort Example

5	2	9	8	4	12	7	1	3	6	10
1	2	3	4	5	6	7	8	9	10	12

1 2 4 5 7 8 9 12 | 3 6 10

2 5 8 9 | 1 4 7 12 | 3 6 10

2 5 | 8 9 | 4 12 | 1 7 | 3 6 | 10

5 | 2 | 9 | 8 | 4 | 12 | 7 | 1 | 3 | 6 | 10

Algorithm BOTTOMUPSORT

Algorithm: ***BOTTOMUPSORT***
Input: An array *A*[1..*n*] of *n* elements.
Output: *A*[1..*n*] sorted in nondecreasing order.

```
t ← 1
while t < n
  s ← t; t ← 2s; i ← 0
  while i + t ≤ n
    MERGE(A, i + 1, i + s, i + t)
    i ← i + t
  end while
  if i + s < n then
    MERGE(A, i + 1, i+ s, n)
end while
```


Analyzing Algorithm BOTTOMUPSORT

- With no loss of generality, assume that the size of the array, n, is a power of 2.
  - In the first iteration, we have pairs that are merged using element comparisons.
  - In the second iteration, we have pairs that are merged using
  - ….
  - In the j^{th} iteration, we have pairs that are merged using
  - The outer while loop is executed times.
  - Therefore,

Analyzing Algorithm BOTTOMUPSORT

- What about the number of element assignments?

Time Complexity

- One way of measuring the performance of an algorithm is how fast it executes. The question is how to measure this “time”?
  - Is having a digital stop watch suitable?
- In general, we are not so much interested in the time and space complexity for small inputs.
- For example, while the difference in time complexity between linear and binary search is meaningless for a sequence with n = 10, it is gigantic for $n = 2^{30}$.

Complexity

•For example, let us assume two algorithms A and B that solve the same class of problems.

•The time complexity of A is 5,000n, the one for B is $\lceil 1.1^n \rceil$ for an input with n elements.

•For n = 10, A requires 50,000 steps, but B only 3, so B seems to be superior to A.

•For n = 1000, however, A requires 5,000,000 steps, while B requires $2.5 \cdot 10^{41}$ steps.

Complexity

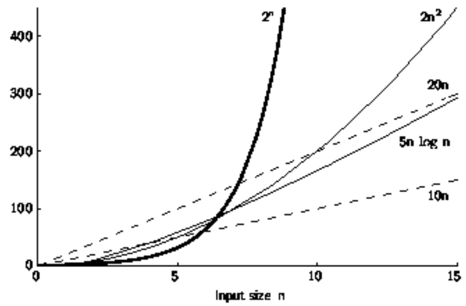
- Comparison: time complexity of algorithms A and B

Input Size	Algorithm A	Algorithm B
n	5,000n	$\lceil 1.1^n \rceil$
10	50,000	3
100	500,000	13,781
1,000	5,000,000	$2.5 \cdot 10^{41}$
1,000,000	$5 \cdot 10^9$	$4.8 \cdot 10^{41392}$

Order of Growth

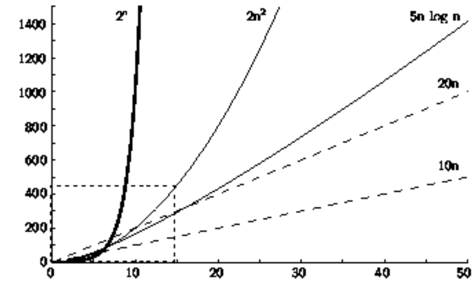
- This means that algorithm B cannot be used for large inputs, while algorithm A is still feasible.
- So what is important is the **growth** of the complexity functions.
- The growth of time and space complexity with increasing input size n is a suitable measure for the comparison of algorithms.
  - we focus on **asymptotic** analysis

Example



Growth rate for some function

Example



Growth rate for same previous functions showing larger input sizes

Running Times for Different Sizes of Inputs of Different Functions

Complexity	10	20	30	40	50	60
n	1×10^{-5} sec	2×10^{-5} sec	3×10^{-5} sec	4×10^{-5} sec	5×10^{-5} sec	6×10^{-5} sec
n^2	0.0001 sec	0.0004 sec	0.0009 sec	0.016 sec	0.025 sec	0.036 sec
n^3	0.001 sec	0.008 sec	0.027 sec	0.064 sec	0.125 sec	0.216 sec
n^5	0.1 sec	3.2 sec	24.3 sec	1.7 min	5.2 min	13.0 min
2^n	0.001sec	1.0 sec	17.9 min	12.7 days	35.7 years	366 cent
3^n	0.59sec	58 min	6.5 years	3855 cent	2×10^{8}cent	1.3×10^{13}cent
$\log_2 n$	3×10^{-6} sec	4×10^{-6} sec	5×10^{-6} sec	5×10^{-6} sec	6×10^{-6} sec	6×10^{-6} sec
$n \log_2 n$	3×10^{-5} sec	9×10^{-5} sec	0.0001 sec	0.0002 sec	0.0003 sec	0.0004 sec

Asymptotic Analysis: Big-oh (O())

- **Definition**: For **T**(n) a non-negatively valued function, **T**(n) is in the set O($f(n)$) if there exist two positive constants c and n_0 such that **T**(n) $\leq cf(n)$ for all $n > n_0$.
- Usage: The algorithm is in O(n^2) in [best, average, worst] case.
- Meaning: For all data sets big enough (i.e., $n > n_0$), the algorithm always executes in less than or equal to $cf(n)$ steps in [best, average, worst] case.

The Growth of Functions

The idea behind the big-O notation is to establish an **upper boundary** for the growth of a function f(x) for large x.

This boundary is specified by a function g(x) that is usually much **simpler** than f(x).

We accept the constant C in the requirement $f(x) \leq C \cdot g(x)$ whenever $x > k$, because **C does not grow with x.**

We are only interested in large x, so it is OK if $f(x) > C \cdot g(x)$ for $x \leq k$.

The Growth of Functions

Example:

Show that $f(x) = x^2 + 2x + 1$ is $O(x^2)$.

For $x > 1$ we have:

$x^2 + 2x + 1 \leq x^2 + 2x^2 + x^2$

$\Rightarrow x^2 + 2x + 1 \leq 4x^2$

Therefore, for C = 4 and k = 1:

$f(x) \leq Cx^2$ whenever $x > k$.

$\Rightarrow f(x)$ is $O(x^2)$.

The Growth of Functions

Question: If f(x) is $O(x^2)$, is it also $O(x^3)$?

Yes. x^3 grows faster than x^2, so x^3 grows also faster than f(x).

Therefore, we always try to find the **smallest** simple function g(x) for which f(x) is O(g(x)).

The Growth of Functions

"Popular" functions g(n) are

n log n, 1, 2^n, n^2, n!, n, n^3, log n

Listed from slowest to fastest growth:

1
log n
n
n log n
n^2
n^3
2^n
n!

The Growth of Functions

A problem that can be solved with polynomial worst-case complexity is called **tractable**.

Problems of higher complexity are called **intractable**.

Later on NP-completeness.

Problems that no algorithm can solve are called **unsolvable**.

Big O() Examples

- Example 1: Find c and n_0 to show that $T(n) = (n+2)/2$ is in $O(n)$
- Example 2: Find c and n_0 to show that $T(n)=c_1n^2+c_2n$ is in $O(n^2)$
- Example 3: $\mathbf{T}(n) = c$. We say this is in O(1).

A **Procedure** to show that f(x) is *O*(g(x))

Show that $3x^3 + 5x^2 - 9 = O(x^3)$.

Let $C = 5$. Let's find k so that $3x^3 + 5x^2 - 9 \le 5x^3$ for $x > k$:

1. Collect terms: $5x^2 \le 2x^3 + 9$
2. What k will make $5x^2 \le x^3$ past k ?
3. $k = 5$!
4. So for $x > 5$, $5x^2 \le x^3 \le 2x^3 + 9$
5. Solution: $C = 5$, $k = 5$ (not unique!)

Asymptotic Analysis: Big-Omega (Ω())

- **Definition**: For $\mathbf{T}(n)$ a non-negatively valued function, $\mathbf{T}(n)$ is in the set $\Omega(g(n))$ if there exist two positive constants c and n_0 such that $\mathbf{T}(n) >= cg(n)$ for all $n > n_0$.
- Meaning: For all data sets big enough (i.e., $n > n_0$), the algorithm always executes in more than or equal to $cg(n)$ steps.
- Ω() notation indicates a lower bound.

Ω() Example

- Find c and n_0 to show that $\mathbf{T}(n) = c_1n^2 + c_2n$ is in $\Omega(n^2)$.

Asymptotic Analysis: Big Theta (Θ())

- When O() and Ω() meet, we indicate this by using Θ() (big-Theta) notation.
- **Definition**: An algorithm is said to be $\Theta(h(n))$ if it is in $O(h(n))$ and it is in $\Omega(h(n))$.

Example

- Show that log(n!) is in $\Theta(n \log n)$.

Complexity Classes and small-oh (*o*())

- Using Θ() notation, one can divide the functions into different equivalence classes, where *f(n)* and *g(n)* belong to the same equivalence class if $f(n) = \Theta(g(n))$
- To show that two functions belong to different equivalence classes, the small-oh notation has been introduced
- **Definition**: Let *f(n)* and *g(n)* be two functions from the set of natural numbers to the set of non-negative real numbers. *f(n)* is said to be in $o(g(n))$ if for every constant $c > 0$, there is a positive integer n_0 such that $f(n) < cg(n)$ for all $n \ge n_0$.

Simplifying Rules

- If $f(n)$ is in $O(g(n))$ and $g(n)$ is in $O(h(n))$, then $f(n)$ is in $O(h(n))$
- If $f(n)$ is in $O(kg(n))$ for any constant $k > 0$, then $f(n)$ is in ....
- If $f_1(n)$ is in $O(g_1(n))$ and $f_2(n)$ is in $O(g_2(n))$, then $(f_1 + f_2)(n)$ is in ....
- If $f_1(n)$ is in $O(g_1(n))$ and $f_2(n)$ is in $O(g_2(n))$ then $f_1(n)f_2(n)$ is in ....
  - You can safely "globally" replace O with Ω or Θ in the above, where the above rules will still hold.

Very Useful Simplifying Rule

- Let *f(n)* and *g(n)* be be two functions from the set of natural numbers to the set of non-negative real numbers such that:

$$0 \le L = \lim_{n\to\infty} \frac{f(n)}{g(n)} \le \infty$$

Then

if L < ∞ then *f(n)* is in $O(g(n))$

if L > 0 then *f(n)* is in $\Omega(g(n))$

if 0 < L < ∞ then *f(n)* is in $\Theta(g(n))$

Big-*O*. Negative Example

$x^4 \neq O(3x^3 + 5x^2 - 9)$:

Show that no *C*, *k* can exist such that past *k*, $C(3x^3 + 5x^2 - 9) \ge x^4$ is always true.

Easiest way is with limits (yes Calculus is good to know):

$$\lim_{x\to\infty} \frac{x^4}{C(3x^3+5x^2-9)} = \lim_{x\to\infty} \frac{x}{C(3+5/x - 9/x^3)}$$

$$= \lim_{x\to\infty} \frac{x}{C(3+0-0)} = \frac{1}{3C} \cdot \lim_{x\to\infty} x = \infty$$

Thus no-matter *C*, x^4 will always catch up and eclipse $C(3x^3 + 5x^2 - 9)$ □

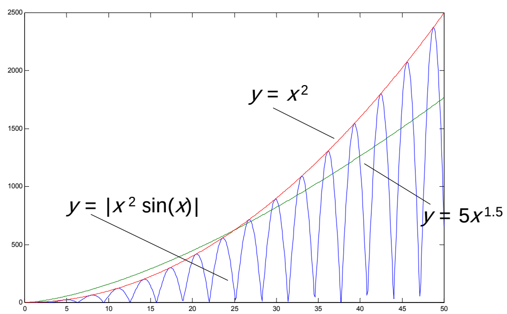
Incomparable Functions

Given two functions *f* (*x*) and *g* (*x*) it is not always the case that one dominates the other so that *f* and *g* are asymptotically *incomparable*.

E.G:

$f(x) = |x^2 \sin(x)|$ vs. $g(x) = 5x^{1.5}$

Incomparable Functions



Space Complexity

- Space complexity refers to the number of memory cells needed to carry out the computational steps required in an algorithm **excluding** memory cells needed to hold the input.
- Compare additional space needed to carry out SELECTIONSORT to that of BOTTOMUPSORT if we have an array with 2 million elements!

Examples

- What is the space complexity for
  - Linear search
  - Binary search
  - Selection sort
  - Insertion sort
  - Merge (that merges two sorted lists)
  - Bottom up merge sort

Estimating the Running Time of an Algorithm

- As mentioned earlier, we need to focus on counting those operations which represent, in general, the behavior of the algorithm
- This is achieved by
  - Counting the frequency of **basic operations**.
    - Basic operation is an operation with highest frequency to within a constant factor among all other elementary operations
  - Recurrence Relations

Counting the Frequency of Basic Operations

- Sometimes, it is easier to compute the frequency of an operation that is a good representative of the overall time complexity of the algorithm
  - For example, Algorithm MERGE.
- Counting the number of iterations
  - The number of iterations in a while loop and/or a for loop is a good indication of the total number of operations

Nested loops

- Running time of a loop equals running time of the code within the loop times the number of iterations.
- Nested Loops: analyze inside out

```
1  for (i=0; i <n; i++)
2      for (j = 0; j< n; j++)
3          k++
```

- Running time of lines 2-3: O(n)
- Running time of lines 1-3: $O(n^2)$

Consecutive statements

- For a sequence S1, S2, .., Sk of statements, running time is maximum of running times of individual statements

  for (i=0; i<n; i++)
      x[i] = 0;
  for (i=0; i<n; i++)
      for (j=0; j<n; j++)
          k[i] += i+j;

- Running time is: $O(n^2)$

Conditional statements

- The running time of

  If (cond) S1

  else S2

  is running time of *cond* plus the max of running times of S1 and S2.

More nested loops

```
1  int k = 0;
2  for (i=0; i<n; i++)
3     for (j=i; j<n; j++)
4        k++
```

- Running time of lines 3-4: n-i
- Running time of lines 1-4:

$$\sum_{i=0}^{n-1}(n-i)=n(n-1)/2=O(n^2)$$

More nested loops

```
1  int k = 0;
2  for (i=1; i<n; i*= 2)
3     for (j=1; j<n; j++)
4        k++
```

- Running time of inner loop: O(n)
- What about the outer loop?
- In m-th iteration, value of i is 2^{m-1}
- Suppose $2^{q-1} < n \le 2^q$, then outer loop is executed q times.
- Running time is O(n log n). Why?

A more intricate example

```
1  int k = 0;
2  for (i=1; i<n; i*= 2)
3     for (j=1; j<i; j++)
4        k++
```

- Running time of inner loop: O(i)
- Suppose $2^{q-1} < n \le 2^q$, then the total running time:

  $1 + 2 + 4 + \ldots + 2^{q-1} = 2^q - 1$

- Running time is O(n).

Computing Fibonacci numbers

- We write the following program: a recursive program

```
1  long int fib(n) {
2  if (n <= 1)
3      return 1;
4  else return fib(n-1) + fib(n-2)
```

- Try fib(100), and it takes forever.
- Let us analyze the running time.

fib(n) runs in exponential time

- The number of operations can be represented as a recurrence relation.
- Let T denote the running time.

  $T(0) = T(1) = c$

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

  where 2 accounts for line 2 plus the addition at line 3.

- By induction, we can show that the running time is $\Omega((3/2)^n)$.
- So the running time grows exponentially.

Efficient Fibnacci numbers

- Avoid recomputation
- Solution with linear running time

```
int fib(int n)
{
   int fibn=0, fibn1=0, fibn2=1;

   if (n < 2)
       return n
    else
     {
       for( int i = 2; i <= n; i++ ) {
          fibn = fibn1 + fibn2;
          fibn1 = fibn2;
          fibn2 = fibn;
       }
      return fibn;
     }
}
```


Average Case Analysis

- Probabilities of all inputs is an important piece of prior knowledge in order to compute the number of operations on average
- Usually, average case analysis is lengthy and complicated, even with simplifying assumptions.

Computing the Average Running Time

- The running time in this case is taken to be the average time over all inputs of size n.
  - Assume we have k inputs, where each input costs C_i operations, and each input can occur with probability P_i, $1 \le i \le k$, the average running time is given by

$$\sum_{i=1}^{k} P_i C_i$$

Average Case Analysis of Linear Search

- Assume that the probability that key x appears in any position in the array (1, 2, …, n) or does not appear in the array is equally likely
  - This means that we have a total of ……… different inputs, each with probability ………
  - What is the number of comparisons for each input?
  - Therefore, the average running time of linear search = ………

Average Case Analysis of Insertion Sort

- Assume that array A contains the numbers from 1..n (i.e. elements are distinct)
- Assume that all n! permutations of the input are equally likely.
- What is the number of comparisons for inserting A[i] in its proper position in A[1..i]? What about on average?
- Therefore, the total number of comparisons on average is

Amortized Analysis

- *The problem*:
  - We have a data structure
  - We perform a sequence of operations
    - Operations may be of different types (e.g., *insert*, *delete*)
    - Depending on the state of the structure the actual cost of an operation may differ (e.g., *inserting into a sorted array*)
  - Just analyzing the worst-case time of a single operation may not say too much
  - We want the average running time of an operation (*but from the worst-case sequence of operations*!).

Binary counter example

- *Example data structure: a binary counter*
  - Operation: *Increment*
  - Implementation: An array of bits $A[0..k–1]$

```
Increment(A)
i ← 0
while i < k and A[i] = 1 do
  A[i] ← 0
  i ← i + 1
if i < k then A[i] ← 1
```

- *How many bit assignments do we have to do in the **worst-case** to perform Increment(A)?* **k-1**
  - But usually we do much less bit assignments!

Analysis of binary counter

- *How many bit-assignments do we do on average*?
  - Let's consider a sequence of *n Increment's*
  - Let's compute the sum of bit assignments:
    - *A*[0] assigned on each operation: *n* assignments
    - *A*[1] assigned every two operations: *n*/2 assignments
    - *A*[2] assigned every four ops: n/4 assignments
    - *A*[*i*] assigned every 2^i ops: $n/2^i$ assignments

$$\sum_{i=0}^{\lfloor \lg n \rfloor} \left\lfloor \frac{n}{2^i} \right\rfloor < 2n$$

- Thus, a single operation takes 2n/n = 2 = O(1) time **amortized** time

Amortized analysis

- Unlike average case analysis, we do not need any probability assumptions
- We compute the average cost per operation for any mix of n operations
- Three techniques for amortization:

1. **Aggregate** - the total amount of time needed for the *n* operations is added and divided by *n*.
2. **Accounting** - operations are assigned an amortized (invented) cost.
   - Usually some of the operations have a *cost* of "0".
   - The rest have a *positive cost*, and "pay" for the "0" cost operations.
3. **Potential function method** not discussed.

Aggregate analysis

- ***Aggregate** analysis – a simple way to do amortized analysis*
  - Treat all operations equally
  - Compute the *worst-case* running time of a sequence of *n* operations.
  - Divide by *n* to get an amortized running time
- Used this method earlier on binary counter

Aggregate analysis – stack example

- *Example data structure: a stack*
  - 3 Operations: *Push, Pop, ClearStack*

ClearStack(S)
while not *empty*(S) **do**
  pop(S)
end while

- **Assume** a sequence of n push, pop and clearStack operations
- In the **worst-case** an operation takes n-1 steps
  - But usually much less!

Aggregate analysis – stack example

- *Push* and *Pop* cost 1
- *ClearStack* costs s where *s* is the size of the stack.
- The number of pushes is at most n
- Each object can be popped only once for each time it is pushed
- So the total number of times *pop* can be called (directly or by *clearStack*) is bound by the number of *push*es ≤ *n*.
- Worst case in n operations total n-1 *push*es and 1 *clearStack*, costing 2(n-1)=2n-2
- The amortized cost of each operation is (2*n*-2)/ *n* ≈ 2, or O(1)

Aggregate analysis – stack example

Operation	Stack	Operation	Stack
Start	(empty)		(empty)
push a	a		a
push b	b, a		b, a
push c	c, b, a		c, b, a
pop c	b, a	clearStack c	(empty)
pop b	a		
pop a	(empty)		
6 Operation: 6 Moves		4 Operation: 6 Moves	

Amortization: Accounting Method

- The **accounting method** determines the amortized running time with a system of credits and debits.
- We view a computer as a coin-operated device requiring 1 unit of cyber-money for a constant amount of computing.
- We set up a scheme for charging operations. This is known as an **amortization scheme**.
  - We may assign different charges to different operations - sometimes more than appropriate, sometimes less.
  - When charged more than the actual cost, an operation will save some credit; when charged less, it will have to draw down some of the accumulated credit.
- The scheme must give us always enough money to pay for the actual cost of the operation – **no negative balance**.
- (amortized time) ≤ (total $ charged) / (# operations)

Accounting Method: Binary counter

- To assign a bit, we have to use one riyal
- When we assign "1", I use one riyal, and we put one riyal in a "savings account".
- When we assign "0", we can do it using a riyal from the savings account.
- *How much do we have to pay for the* Increment(*A*) *for this scheme to work* if the counter starts with 0?
  - Only one assignment of "1" in the algorithm. Obviously, two riyals will always pay for the operation
- We assign the amortized costs:
  - **SR2 for 0→1 flip** and **SR0 for 1→0 flip**
- With these costs, balance is always nonnegative. **Why?**

Accounting Method: Stack Example

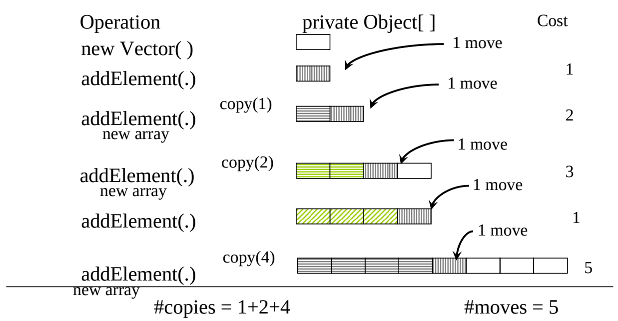
- We assign the amortized costs:
  - **SR2 for *push***
  - **SR0 for both *pop* and *clearStack***
- For a sequence of *n push*, *pop* and *clearStack* operations the cost is at most SR2*n* (i.e. max *n* pushes.)
- Each time we do a *push* we pay SR1 for the cost of the *push* and the element has a credit of SR1.
- Each time an element is popped we take SR1 from the element to pay for it.
- Since the balance is never negative, amortized costs of SR2 for *push* and SR0 for *pop* and *clearStack* satisfy the balance constraint.

Dynamic Tables

- In an insert operation, when the array is full, instead of throwing an exception, we can replace the array with a larger one
- How large should the new array be?
  - keep it as small as possible
  - incremental strategy: increase the size by a constant *c*
  - doubling strategy: double the size

```
Algorithm insert(o)
  if t = S.length – 1 then
    A ← new array of
        size …
    for i ← 0 to t do
      A[i] ← S[i]
      S ← A
  t ← t + 1
  S[t] ← o
```

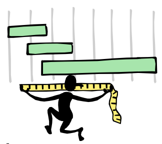

java.util.Vector - addElement()



#copies = 1+2+4 #moves = 5

Total # = (1+2+4) + 5

Comparison of the Strategies



- We compare the incremental strategy and the doubling strategy by analyzing the total time $T(n)$ needed to perform a series of n insert operations
- We assume that we start with an empty table represented by an array of size 1

Analysis of the Incremental Strategy

- We replace the array $k = \lfloor n/c \rfloor$ times
- The total time $T(n)$ of a series of n insert operations is proportional to

$$n + c + 2c + 3c + 4c + \dots + kc =$$
$$n + c(1 + 2 + 3 + \dots + k) =$$
$$n + ck(k + 1)/2$$

- Since c is a constant, $T(n)$ is $O(n + k^2)$, i.e., $O(n^2)$
- The amortized time of an insert operation is $O(n)$

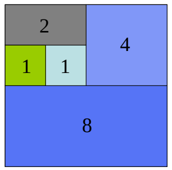
Aggregate Analysis of the Doubling Strategy

- We replace the array $k = \lceil \log_2 n \rceil$ times
- The total time $T(n)$ of a series of n insert operations is proportional to

  $n + 1 + 2 + 4 + 8 + \ldots + 2^{k-1} =$

  $n + 2^k - 1 = 3n - 1$
- $T(n)$ is $O(n)$
- The amortized time of an insert operation is $O(1)$

geometric series

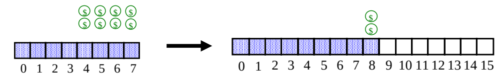


Accounting Analysis of Doubling Strategy

- Charge each operation SR3 amortized cost
  - Use SR1 to perform immediate Insert()
  - Save SR2
- When table doubles
  - SR2 reinserts old item
  - Point is, we've already paid these costs
  - Upshot: **constant (amortized) cost per operation**

Amortization Scheme for the Doubling Strategy

- Consider again the k phases, where each phase consisting of twice as many insert s as the one before.
- At the end of a phase we must have saved enough to pay for the array-growing insert of the next phase.
- At the end of phase i we want to have saved 2^i cyber-riyals, to pay for the array growth for the beginning of the next phase.
- We charge SR3 for a push. The SR2 saved for a regular push are "stored" in the second half of the array.



Algorithm for Surjectivity

Q: Why is the second algorithm better than the first?

A: Because the second algorithm runs faster. Even under the criterion of code-length, algorithm 2 is better.

Let's see why:

Running time of 1st algorithm

```
boolean isOnto( function f: (1,
   2,…, n) → (1, 2,…, m) ){
   if( m > n ) return false          1 step OR:
   soFarIsOnto = true                1 step (assigment)
   for( j = 1 to m ){                m loops: 1 increment plus
     soFarIsOnto = false               1 step (assignment)
     for(i = 1 to n ){                 n loops: 1 increment plus
        if ( f(i ) == j )                1 step possibly leads to:
          soFarIsOnto = true                1 step (assignment)
        if( !soFarIsOnto )               1 step possibly leads to:
          return false                      1 step (return)
     }
   }
   return true;                      possibly 1 step
}
```


Running time of 1st algorithm

```
                                   WORST-CASE running time:
                                   Number of steps = 1 OR 1+
1 step (m>n) OR:                     1 +
1 step (assigment)                   m ·
m loops: 1 increment plus            (1+ 1 +
  1 step (assignment)                  n ·
  n loops: 1 increment plus              (1+1
    1 step possibly leads to:              + 1
      1 step (assignment)                 + 1
    1 step possibly leads to:              + 1
      1 step (return)                      )
possibly 1 step                        + 1
                                     )
                                   = 1 (if m>n) OR 5mn+3m+2
```


Running time of 2nd algorithm

```
boolean isOntoB( function f: (1,
   2,…, n) → (1, 2,…, m) ){
    if( m > n ) return false         1 step OR:
   for( j = 1 to m )                 m loops: 1 increment plus
        beenHit[ j ] = false               1 step (assignment)
   for(i = 1 to n )                  n loops: 1 increment plus
        beenHit[ f(i ) ] = true            1 step (assignment)
   for(j = 1 to m )                  m loops: 1 increment plus
        if( !beenHit[ j ] )                1 step possibly leads to:
                return false                  1 step
   return true                       possibly 1 step
}
```


Running time of 2nd algorithm

```
                                   WORST-CASE running time:
1 step (m>n) OR:                   Number of steps = 1 OR 1+
m loops: 1 increment plus            m · (1+
     1 step (assignment)               1)
n loops: 1 increment plus          + n · (1+
     1 step (assignment)               1 )
m loops: 1 increment plus          + m · (1+
     1 step possibly leads to:         1
        1 step                         + 1)
possibly 1 step                    + 1
                                   = 1 (if m>n) OR 5m + 2n + 2
```


Comparing Running Times

The first algorithm requires at most $5mn+3m+2$ steps, while the second algorithm requires at most $5m+2n+2$ steps. In both cases, for *worst case* times we can assume that $m \leq n$ as this is the longer-running case (for the other case, constant time). This reduces the respective running times to $5n^2+3n+2$ and $5n+2n+2= 8n+2$.

To tell which algorithm is better, find the most important terms using big-Θ notation:

- $5n^2+3n+2 = \Theta(n^2)$ –***quadratic*** time complexity
- $8n+2 = \Theta(n)$ –***linear*** time complexity ***WINNER***

Q: Any issues with this line of reasoning?

Comparing Running Times. Issues

1. Inaccurate to summarize running times $5n^2+3n+2$, $8n+2$ only by biggest term. For example, for n=1 both algorithms take 10 steps.
2. Inaccurate to count the number of "basic steps" without measuring how long each basic step takes. Maybe the basic steps of the second algorithm are much longer than those of the first algorithm so that in actuality first algorithm is faster.

Comparing Running Times. Issues

3. Surely the running time depends on the platform on which it is executed. E.g., C-code on a Pentium IV will execute much faster than Java on a Palm-Pilot.
4. The running time calculations counted many operations that may not occur. In fact, a close look reveals that we can be certain the calculations were an over-estimate since certain conditional statements were mutually exclusive. Perhaps we over-estimated so much that algorithm 1 was actually a *linear-time* algorithm.

Comparing Running Times. Responses

1. **Big-Θ inaccurate**: Quadratic time Cn^2 will always take longer than linear time Dn for large enough input, no matter what C and D are; furthermore, it is the large input sizes that give us the real problems so are of most concern.

Comparing Running Times. Responses

2. "Basic steps" counting inaccurate: True that we have to define what a basic step is.

EG: Does multiplying numbers constitute a basic step or not. Depending on the computing platform, and the type of problem (e.g. multiplying `int`'s vs. multiplying arbitrary integers) multiplication may take a fixed amount of time, or not. When this is ambiguous, you'll be told explicitly what a basic step is.

Q: What were the ***basic steps*** in previous algorithms?

Comparing Running Times

A: Basic steps—

Assignment	Increment
Comparison	Negation
Return	Random array access
Function output access	

Each may in fact require a different number ***bit operations*** –the actual operations that can be carried out in a single cycle on a processor. However, since each operation is itself $O(1)$ --i.e. takes a constant amount of time, asymptotically as if each step was in fact 1 time-unit long!

Comparing Running Times. Issues

3. Platform dependence: Turns out there is usually a constant multiple between the various basic operations in one platform and another. Thus, big-O erases this difference as well.
4. Running time is too pessimistic: It is definitely true that when $m > n$ the estimates are over-kill. Even when m=n there are cases which run much faster than the big-Theta estimate. However, since we can always find inputs which do achieve the big-Theta estimates (e.g. when f is onto), and the worst-case running time is defined in terms of the worst possible inputs, the estimates are valid.

Worst Case vs. Average Case

The time complexity described above is ***worst case*** complexity. This kind of complexity is useful when one needs absolute guarantees for how long a program will run. The worst case complexity for a given n is computed from the case of size n that takes the longest.

On other hand, if a method needs to be run repeatedly many times, ***average case*** complexity is most suitable. The average case complexity is the avg. complexity over all possible inputs of a given size.

Usually computing avg. case complexity requires probability theory.

Q: Does one of the two surjectivity algorithms perform better on average than worst case?

Worst Case vs. Average Case

A: Yes. The first algorithm performs better on average. This is because surjective functions are actually rather rare, and the algorithm terminates early when a non-hit element is found near the beginning.

With probability theory will be able to show that when $m = n$, the first algorithm has $O(n)$ average complexity.

Big-O A Grain of Salt

Big-O notation gives a good first guess for deciding which algorithms are faster. In practice, the guess isn't always correct.

Consider time functions n^6 vs. $1000n^{5.9}$. Asymptotically, the second is better. Often catch such examples of purported advances in theoretical computer science publications. The following graph shows the relative performance of the two algorithms:

Big-*O*
A Grain of Salt