Chap 2 – Getting Started

2.1 Insertion sort

2.2 Analyzing algorithms

2.3 Designing algorithms

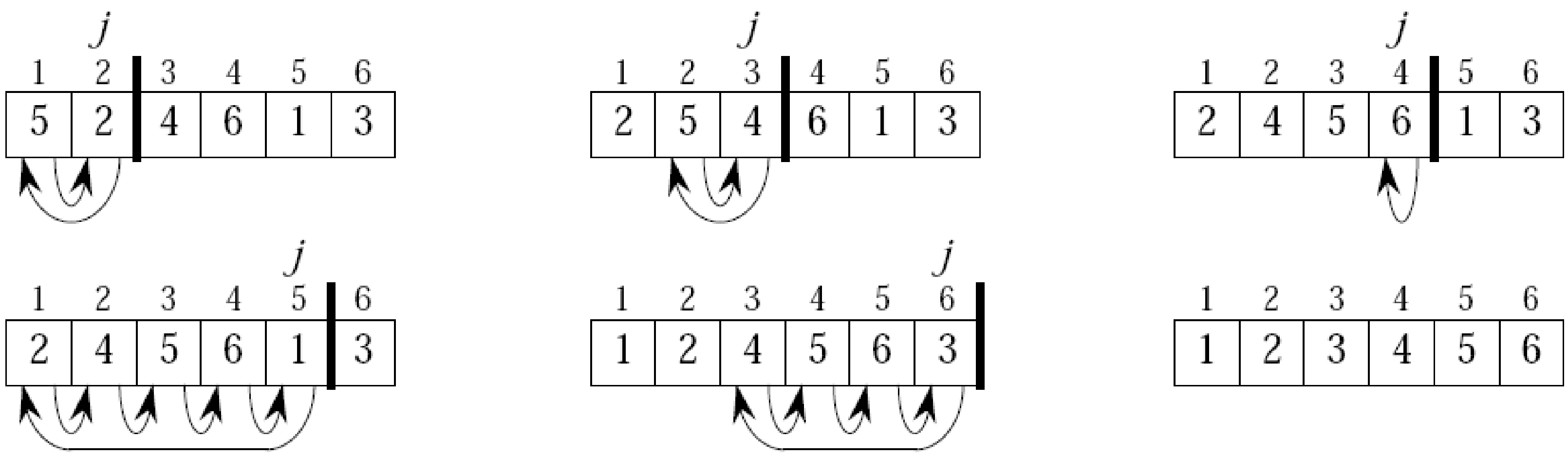
2.1 Insertion sort

- The sorting problem

  Input: A sequence of n numbers $\langle a_1, a_2, \dots, a_n \rangle$.

  Output: A permutation $\langle a'_1, a'_2, \dots, a'_n \rangle$ of the input sequence such that $a'_1 \le a'_2 \le \cdots \le a'_n$

- Insertion sort



2.1 Insertion sort

- Insertion sort

```
INSERTION-SORT(A, n)
for j = 2 to n
    key = A[j]
    // insert A[j] into the sorted sequence A[1..j − 1]
    i = j − 1
    while i > 0 and A[i] > key
        A[i + 1] = A[i]
        i = i − 1
    A[i + 1] = key
```

2.1 Insertion sort

- Correctness of insertion sort
  - Loop invariant (for the **for** loop)

    At line 1, $A[1..j-1]$ is a permutation of the elements originally in $A[1..j-1]$, but in sorted order.
  - Initialization: $j=2$, trivial
  - Maintenance

    Clearly, the loop invariant remains true after inserting $A[j]$ into $A[1..j-1]$.

    (A formal proof needs a loop invariant for the **while** loop.)
  - Termination: $j=n+1$

    The entire array $A[1..n]$ is a permutation of the elements originally in $A[1..n]$, but in sorted order.

2.2 Analyzing algorithms

- Computation model
  - Algorithms are implemented and analyzed within a computation model.
  - Random-access machine (RAM): a model of modern single-processor machine
- Input size
  - The time taken by an algorithm depends on the input size.
  - Input size depends on the problem being studied.

    Sorting problem: the number of elements being sorted

    Number problem: the number of bits

    Graph problem: the number of vertices and edges

2.2 Analyzing algorithms

- Analysis of insertion sort

INSERTION-SORT(A, n)	cost	times
for $j = 2$ **to** n	c_1	n
$key = A[j]$	c_2	$n - 1$
// insert $A[j]$ into $A[1..j-1]$	0	$n - 1$
$i = j - 1$	c_4	$n - 1$
while $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
$A[i+1] = A[i]$	c_6	$\sum_{j=2}^{n} (t_j - 1)$
$i = i - 1$	c_7	$\sum_{j=2}^{n} (t_j - 1)$
$A[i+1] = key$	c_8	$n - 1$

t_j = the number of times the test $i > 0$ and $A[i] > key$ is executed for the value of j. Note that $1 \le t_j \le j$

2.2 Analyzing algorithms

- Best case
  - The array is already sorted.
  - $t_j = 1,\ \sum_{j=2}^{n} t_j = n - 1,\ \ \sum_{j=2}^{n}(t_j - 1) = 0$
  - Let $T(n)$ = the best case running time taken by insertion sort on n elements. Then,

    $T(n)$
    $= c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1)$
    $= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8)$
    $= an + b$
  - Hardly useful

2.2 Analyzing algorithms

- Worst case
  - The array is in reverse sorted order.
  - $t_j = j$, $\sum_{j=2}^{n} t_j = \frac{n(n+1)}{2} - 1$, $\sum_{j=2}^{n} (t_j - 1) = \frac{n(n-1)}{2}$
  - Let $T(n)$ = the worst case running time taken by insertion sort on n elements. Then,

$$
\begin{aligned}
T(n) & \\
&= c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) \\
&\quad + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1) \\
&= an^2 + bn + c
\end{aligned}
$$

2.2 Analyzing algorithms

- Average case
  - Assume that the n elements are distinct and each of the $n!$ possible inputs is equally likely.

$A[1]$	$\cdots$	$A[i-1]$	$A[i]$	$\cdots$	$A[j-1]$	$A[j]$
1	$\cdots$	$i-1$	$i+1$	$\cdots$	j	i

$$t_j = \sum_{i=1}^{j} \Pr[A[j] \text{ is the } i\text{th smallest}] \times (j-i+1)$$

$$= \sum_{i=1}^{j} \frac{1}{j} \times (j-i+1) = \frac{1}{j} \sum_{i=1}^{j} i = \frac{j+1}{2}$$

2.2 Analyzing algorithms

- Average case
  - Let $T(n)$ = the average case running time taken by insertion sort on n elements

    Then, $T(n) \approx \frac{1}{2} \cdot$ worst-case running time
  - Less interesting than the worst case analysis
    - The worst case gives a guaranteed upper bound
    - Depend on probabilistic assumption
    - Often as bad as the worst case in terms of order of growth

2.2 Analyzing algorithms

- Order of growth
  - An abstraction to ease analysis and focus on important features.
  - Look only at the leading term
    - Drop the lower-order terms
      They are less significant than the higher-order terms.
      e.g. $n^2 + 3n + 10 \quad n = 1000,\ 1000000 + 3000 + 10$
    - Ignore the constant coefficient in the leading term
      It is less significant than the rate of growth for large inputs.
      e.g. $2n^2 \rightarrow 3n^2 \quad n = 100,\ \ 20000 \rightarrow 30000$
      $2n^2 \rightarrow 2n^3 \quad n = 100,\ \ 20000 \rightarrow 2000000$

2.2 Analyzing algorithms

- Order of growth
  - E.g. Insertion sort

    $T(n) = an^2 + bn + c = \Theta(n^2)$

    We say that $T(n)$ grows like n^2, rather than $T(n)$ equals n^2
- Analyzing basic operations
  - Since the order of growth is concerned, only the basic operation needs counting.
  - E.g. For insertion sort, the comparison between array elements is the basic operation.

    In the worst case, the number of times it is executed is

    $\sum_{j=2}^{n}(j-1) = \frac{n(n-1)}{2} = \Theta(n^2)$

2.3 Designing algorithms

- Algorithm design
  - Incremental – insertion sort
  - Divide and conquer – merge sort
- Merge sort

MERGE-SORT(A, p, r)

if $p < r$

$\quad q = \lfloor (p + r)/2 \rfloor$

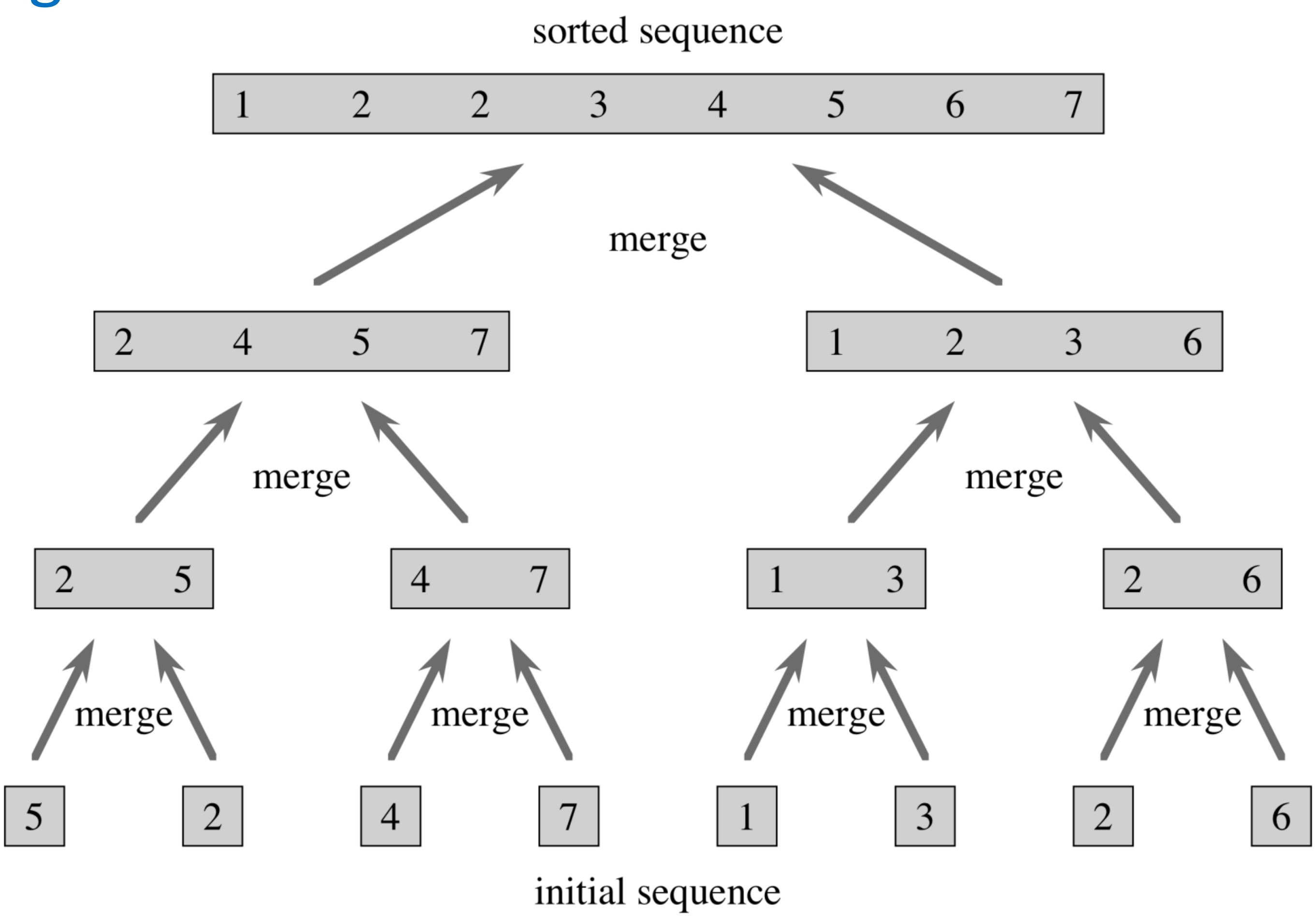
$\quad$ MERGE-SORT(A, p, q)

$\quad$ MERGE-SORT$(A, q + 1, r)$

$\quad$ MERGE(A, p, q, r)

2.3 Designing algorithms

- Merge sort



2.3 Designing algorithms

- Linear-time merge

MERGE(A, p, q, r)

$n_1 = q - p + 1$

$n_2 = r - q$

Let $L[1..n_1 + 1]$ and $R[1..n_2 + 1]$ be new arrays

$$\left.\begin{array}{l}\textbf{for } i = 1 \textbf{ to } n_1 \textbf{ do } L[i] = A[p + i - 1] \\ \textbf{for } j = 1 \textbf{ to } n_2 \textbf{ do } R[j] = A[q + j]\end{array}\right\} \Theta(n)$$

$L[n_1 + 1] = R[n_2 + 1] = \infty$ // sentinel

$i = j = 1$ $n = n_1 + n_2$

$$\left.\begin{array}{l}\textbf{for } k = p \textbf{ to } r \\ \quad \textbf{if } L[i] \le R[j] \textbf{ then } A[k] = L[i]; i = i + 1 \\ \quad \textbf{else } A[k] = R[j]; j = j + 1\end{array}\right\} \Theta(n)$$

2.3 Designing algorithms

- Analysis of merge sort
  - Let $T(n)$ = the (worst case) running time taken by merge sort on n elements. Then,

    $$T(n) = \begin{cases} \Theta(1) & n = 1 \\ 2T(n/2) + \Theta(n) & n > 1 \end{cases}$$

    or

    $$T(n) = \begin{cases} c & n = 1 \\ 2T(n/2) + cn & n > 1 \end{cases}$$

    Why the same constant c?

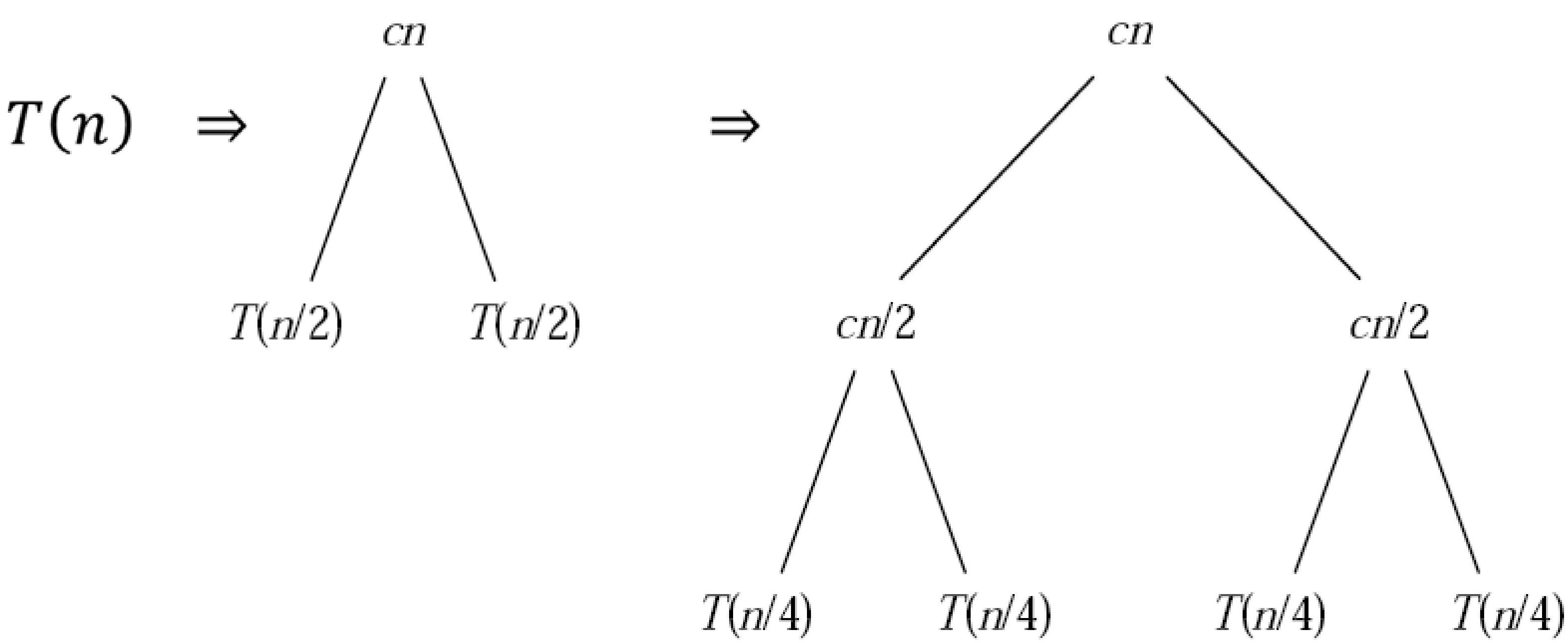
    Let $a = \Theta(1), bn = \Theta(n)$

    Let $c = \max(a, b)$, $T(n) = O(n \lg n)$ is an upper bound

    Let $c = \min(a, b)$, $T(n) = \Omega(n \lg n)$ is a lower bound

2.3 Designing algorithms

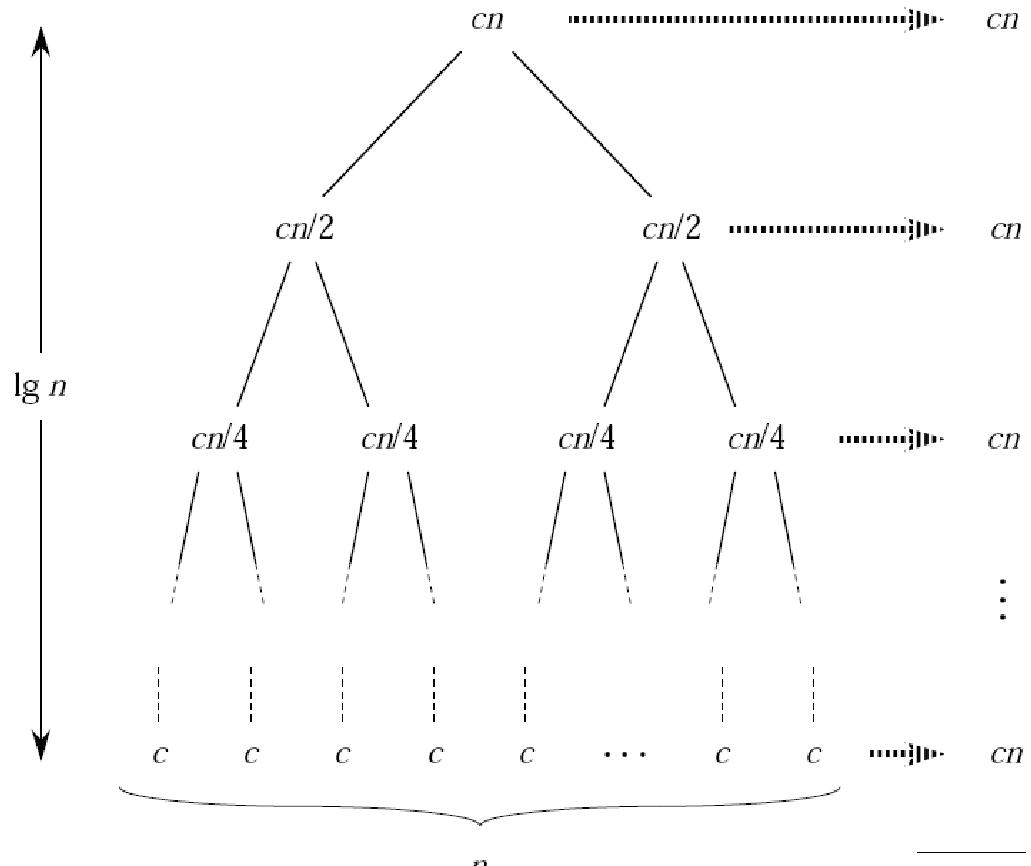
- Recursion tree (for $n = 2^k$)



$$T(n) = \begin{cases} c & n = 1 \\ 2T(n/2) + cn & n > 1 \end{cases}$$

2.3 Designing algorithms

- Recursion tree (for $n = 2^k$)



Total cost: $T(n) = cn(k+1) = cn(\lg n + 1) = \Theta(n \lg n)$

2.3 Designing algorithms

- Analysis of merge sort
  - Another recurrence $T(n) = \begin{cases} a & n = 1 \\ 2T(n/2) + cn & n > 1 \end{cases}$

    Then, $T(n) = cn \lg n + an = \Theta(n \lg n)$
- Comparison between insertion sort and merge sort
  - Space complexity

    Insertion sort – $\Theta(1)$ stack space

    Merge sort – $\Theta(n)$ array space and $\Theta(\lg n)$ stack space
  - Time complexity

    Insertion sort is faster than merge sort on small inputs, as the hidden constant of $\Theta(n^2)$ is smaller than that of $\Theta(n \lg n)$

Experiment

- Profiling insertion sort
  - Step 1 – Code the algorithm correctly
  - Step 2 – Configure the worst case input and remove output

File isort.cpp

```
//#include <iostream>
//using namespace std;
//#define k 20
int main()
{
    int a[k];
    for (int i=0;i<k;i++) a[i]=k-i;
    isort(a,k);
//  for (int i=0;i<k;i++)
//      cout << a[i] << ' ';
}
```

```
void isort(int* a,int n)
{
    for (int j=1;j<n;j++) {
        int key=a[j],i=j-1;
        while (i>=0&&a[i]>key) {
            a[i+1]=a[i]; i--;
        }
        a[i+1]=key;
    }
}
```

Experiment

- Profiling insertion sort
  - Step 3 – Profiling

bsd2> g++ -Dk=10000 -pg isort.cpp

bsd2> ./a.out # yield monitor file a.out.gmon

bsd2> gprof # yield (A) call graph profile (B) flat profile

(A)

…	…	…	…	…	…	…
…	…	…	…	…	…	…

(B)

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
100.0	0.29	0.29	1	292.50	292.50	_Z5isortPii [2]
0.0	0.29	0.00	3	0.00	0.00	stub_zero [4]
…	…	…	…	…	…	…

Experiment

- A shell script for profiling insertion sort

```
g++ -Dk=$1 -pg isort.cpp
for i in 1 2 3
do
    ./a.out
    gprof -b | grep -A2 cumulative > tmp
    if test $i -eq 1
    then
        grep -A1 cumulative tmp
    fi
    grep isort tmp
done
rm tmp
```

File pisort

if not work,
mv a.out.gmon gmon.out
or
gprof a.out a.out.gmon -b

Experiment

- Running the shell script pisort

bsd2> sh pisort 10000

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
100.0	0.29	0.29	1	292.50	292.50	_Z5isortPii [2]
100.0	0.29	0.29	1	293.00	293.00	_Z5isortPii [2]
100.0	0.29	0.29	1	292.50	292.50	_Z5isortPii [2]

bsd2> sh pisort 100000

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
100.0	29.41	29.41	1	29405.00	29405.00	_Z5isortPii [3]
100.0	29.34	29.34	1	29344.00	29344.00	_Z5isortPii [3]
100.0	29.34	29.34	1	29343.00	29343.00	_Z5isortPii [3]

Experiment

- Determine the coefficient hidden in $T(n) = \Theta(n^2)$

  Let

  $T(n) =$ the worst case running time of insertion sort on bsd2

  We have

  $T(10^4) = c \times 10^8 = 0.29 \Rightarrow c \approx 2.9 \times 10^{-9}$

  $T(10^5) = c \times 10^{10} = 29.36 \Rightarrow c \approx 2.9 \times 10^{-9}$

  Thus,

  $T(n) \approx 2.9 \times 10^{-9} n^2$

Experiment

- Profiling merge sort
  - Source code

File msort.cpp

```
//#include <iostream>
//using namespace std;
//#define n 20
int main()
{
        int a[n];
//      for (int i=0;i<n;i++) a[i]=n-i;
        msort(a,0,n-1);
//      for (int i=0;i<n;i++)
//              cout << a[i] << ' ';
}
```

```
void msort(int* a,int l, int h)
{
        if (l < h) {
                int m=(l+h)/2;
                msort(a,l,m);
                msort(a,m+1,h);
                merge(a,l,m,h);
        }
}
```

Experiment

- Profiling merge sort
  - Source code

```
void merge(int* a,int l,int m,int h)
{
    static int b[n];
    int i=l,j=m+1,k=l;
    while (i<=m&&j<=h)
        if (a[i]<a[j]) b[k++]=a[i++];
        else b[k++]=a[j++];
    while (i<=m) b[k++]=a[i++];
    while (j<=h) b[k++]=a[j++];
    for (i=l;i<=h;i++) a[i]=b[i];
}
```

Experiment

- Profiling merge sort
  - bsd2> g++ -Dn=1000000 -pg msort.cpp

    bsd2> ./a.out

    bsd2> gprof -b | grep -A5 cumulative

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
63.6	0.21	0.21	999999	0.00	0.00	_Z5mergePiiii [4]
28.0	0.28	0.09	0	100.00%		_mcount [5]
4.7	0.30	0.01	0	100.00%		.mcount (13)
3.7	0.31	0.01	1	11.50	209.00	_Z5msortPiii [2]

Experiment

- A shell script for profiling merge sort

```
g++ -Dn=$1 -pg msort.cpp
for i in 1 2 3
do
    ./a.out
    gprof -b | grep -A5 cumulative >  tmp
    if test $i -eq 1
    then
        grep -A1 cumulative tmp
    fi
    grep merge tmp
    grep msort tmp
done
rm tmp
```

File pmsort

Experiment

- Running the shell script pmsort

bsd2> sh pmsort 1000000

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
64.5	0.21	0.21	999999	0.00	0.00	_Z5mergePiiii [4]
3.6	0.31	0.01	1	11.50	217.50	_Z5msortPiii [2]
62.2	0.20	0.20	999999	0.00	0.00	_Z5mergePiiii [4]
3.4	0.31	0.01	1	11.00	210.00	_Z5msortPiii [2]
64.2	0.20	0.20	999999	0.00	0.00	_Z5mergePiiii [4]
5.2	0.30	0.02	1	16.00	214.50	_Z5msortPiii [2]

Experiment

- Determine the coefficient hidden in $T(n) = \Theta(n \lg n)$

  Let

  $T(n) =$ the worst case running time of merge sort on bsd2

  We have

  $T(10^6) = c \times 10^6 \lg 10^6 = 0.22 \Rightarrow c \approx 1.1 \times 10^{-8}$

  $T(10^7) = c \times 10^7 \lg 10^7 = 2.44 \Rightarrow c \approx 1.1 \times 10^{-8}$

  Thus,

  $T(n) \approx 1.1 \times 10^{-8} n \lg n$

  N.B. Input size 10 times larger, running time

  $\frac{10n \lg 10n}{n \lg n} = \frac{10(\lg 10 + \lg n)}{\lg n} = 10 + \frac{10 \lg 10}{\lg n}$ times slower

  For $n = 10^6$, about 11.6 times slower

Experiment

- Determine the threshold

  For merge sort to beat insertion sort on bsd2 in the worst case we need

  $2.9 \times 10^{-9} n^2 \geq 1.1 \times 10^{-8} n \lg n$

  $2.9 \times 10^{-9} n^2 \geq 11 \times 10^{-9} n \lg n$

  $2.9n \geq 11 \lg n$

  $n \geq 3.8 \lg n \Rightarrow n \geq 15$

n	$3.8 \lg n$
4	7.6
8	11.4
14	14.5
15	14.8
16	15.2

Experiment

- Mergesort refinement

  To sort array a without copy-back

  Step 1 Copy array a to array b

  Step 2 Sort array b into array a as follows:

$b[0..3] \rightarrow a[0..3]$

merge $b[0..1]$ and $b[2..3]$ to $a[0..3]$

$a[0..1] \rightarrow b[0..1]$

merge $a[0..0]$ and $a[1..1]$ to $b[0..1]$

$a[2..3] \rightarrow b[2..3]$

merge $a[2..2]$ and $a[3..3]$ to $b[2..3]$

$b[0..0] \rightarrow a[0..0]$ $b[1..1] \rightarrow a[1..1]$ $b[2..2] \rightarrow a[2..2]$ $b[3..3] \rightarrow a[3..3]$

The refined merge sort needs only half array copies ($\lg n$ vs $2\lg n$)

Experiment

- Insertion sort refinement
  - Instead of sequential search, use binary search to locate the insertion point.

$l \rightarrow$	1	$l \rightarrow$	1		1		1
	2		2		2	$h \rightarrow$	2
	4	$h \rightarrow$	4	$l,h \rightarrow$	4	$l \rightarrow$	4
	5		5		5		5
	6		6		6		6
	7		7		7		7
$h \rightarrow$	8		8		8		8
	3		3		3		3

It takes about $\lg n$ comparisons to locate the insertion point within n elements.

Experiment

- Insertion sort refinement
  - Complexity with sequential search

    $\Theta(n^2)$ comparisons in the worst case

    $\Theta(n^2)$ data movements in the worst case
  - Complexity with binary search

    $\Theta(n \lg n)$ comparisons (in every case)

    $$\because \Theta\left(\sum_{i=1}^{n-1} \lg i\right) = \Theta(\lg(n-1)!) = \Theta(n \lg n)$$

    $\Theta(n^2)$ data movements in the worst case