CSE 2320 Notes 1: Algorithmic Concepts

(Last updated 1/20/16 10:10 AM)

1.A. A Simple Algorithmic Problem - Maintaining Disjoint Subsets

Sedgewick 1.2 and 1.3

Abstraction:

Set of *n* elements: 0 . . *n* - 1

Initially all elements are in *n* different subsets

`find(i)` - Returns integer ("leader") indicating which subset includes `i`

`i` and `j` are in the same subset ⇔ `find(i)==find(j)`

`union(i,j)` - Takes the set union of the subsets with leaders `i` and `j`.

Implementation 1: (`http://ranger.uta.edu/~weems/NOTES2320/uf1.c`)

Initialization:

```
for (i=0; i<n; i++)
  id[i]=i;
```

```
find(i):

      return id[i];
```

```
unionFunc(i,j):

      for (k=0; k<n; k++)
        if (id[k]==i)
          id[k]=j;
```

0	1	2	3	4
0	1	2	3	4

Implementation 2: (`http://ranger.uta.edu/~weems/NOTES2320/uf2.c`)

```
find(i):

      while (id[i]!=i)
        i=id[i];
      return i;
```

```
unionFunc(i,j):

      id[i]=j;
```

0	1	2	3	4
0	1	2	3	4

Implementation 3: (http://ranger.uta.edu/~weems/NOTES2320/uf3.c)

Initialization:

```
for (i=0; i<n; i++)
{
  id[i]=i;
  sz[i]=1;
}
```

find(i):

```
while (id[i]!=i)
  i=id[i]=id[id[i]];  // =id[id[i]] is optional
return i;
```

unionFunc(i,j):

```
if (sz[i]<sz[j])
{
  id[i]=j;
  sz[j]+=sz[i];
}
else
{
  id[j]=i;
  sz[i]+=sz[j];
}
```

Best-case (shallow tree) and worst-case (deep tree) for a sequence of unions?

1.B. QUADRATIC TIME SORTS:

Selection Sort (Sedgewick 6.2)

```
void selection(Item a[], int ell, int r)
{
  int i, j;
  for (i = ell; i < r; i++)
  {
    int min = i;
    for (j = i+1; j <= r; j++)
      if (less(a[j], a[min]))
        min = j;
    exch(a[i], a[min]);
  }
}
```

Always uses $\sum_{i=2}^{n}(i-1) = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} \approx \frac{n^2}{2}$ comparisons and is **<u>not</u>** *stable*.

Insertion Sort with *Sentinel* for Inner Loop (Sedgewick 6.3,
http://ranger.uta.edu/~weems/NOTES2320/insertionSort.c)

```
void insertion(Item a[], int ell, int r)
{
  int i;
  for (i = ell+1; i <= r; i++) // Ensures a[ell] is instance of
    compexch(a[ell], a[i]);    // smallest value using swaps
  for (i = ell+2; i <= r; i++)
  { // a[ell] ... a[i-1] are in ascending order
    int j = i;
    Item v = a[i];
    while (less(v, a[j-1])) // Rippling to fix prefix
    {
      a[j] = a[j-1];
      j--;
    }
    a[j] = v; // After this assignment a[ell] ... a[i] are ordered
  }
}
```

Maximum ("worst case") number of times that body of `j`-loop executes
for a particular value of `i`?

Maximum number of times that body of `j`-loop executes over entire sort?

$$\sum_{i=1}^{k} i = \frac{k(k+1)}{2} = ?$$

Expected ("average") number of times that body of `j`-loop executes for a particular value of `i`?

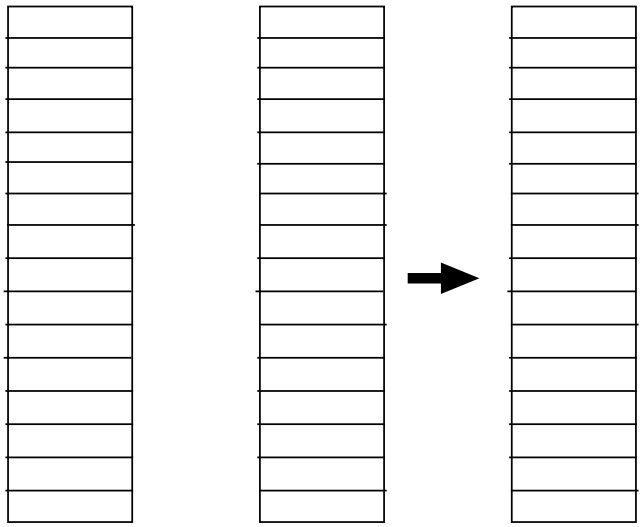
Expected number of times that body of `j`-loop executes over entire sort?

Stable: With sentinel? Without sentinel?

1.C. DIVIDE AND CONQUER (Decomposition)

Sedgewick 5.2, 8.1, 8.3

1. Divide into subproblems (unless size allows a trivial solution).
2. Conquer the subproblems.
3. Combine solutions to subproblems.



```
mergeAB(Item c[], Item a[], int N, Item b[], int M )
{ int i, j, k;
  for (i = 0, j = 0, k = 0; k < N+M; k++)
  {
    if (i == N) { c[k] = b[j++]; continue; }
    if (j == M) { c[k] = a[i++]; continue; }
    c[k] = (less(a[i], b[j])) ? a[i++] : b[j++];
  }
}
```

(http://ranger.uta.edu/~weems/NOTES2320/mergesort.c)

```
mergeAB(Item c[], Item a[], int N, Item b[], int M )
{
  int i = 0, j = 0, k = 0;

  while (i < N && j < M)
    if (less(a[i],b[j]))    // !less(b[j],a[i]) will make mergesort stable
      c[k++] = a[i++];
    else
      c[k++] = b[j++];

  if (i < N)
    for ( ; i < N; i++)
      c[k++] = a[i];
  else
    for ( ; j < M; j++)
      c[k++] = b[j];
}
```

How are items with identical keys (“duplicates”) handled?

Fall 2009 Test Problem Applying Merge Concept

Two `int` arrays, `A` and `B`, contain `m` and `n` `int`s each, respectively. The elements within each of these arrays appear in ascending order without duplication (i.e. each table represents a set). Give Java code for a $\Theta(m+n)$ algorithm to find the **symmetric difference** by producing a third array `C` (in ascending order) with the values that appear in **exactly** one of `A` and `B` **and** sets the variable `p` to the final number of elements copied to `C`. (Details of input/output, allocation, declarations, error checking, comments and style **are unnecessary**.)

```
i=j=p=0;

while (i<m && j<n)
  if (A[i]<B[j])
    C[p++]=A[i++];
  else if (A[i]>B[j])
    C[p++]=B[j++];
  else
  {
    i++;
    j++;
  }

for ( ; i<m; i++)
  C[p++]=A[i];
for ( ; j<n; j++)
  C[p++]=B[j];
```

(Binary) Mergesort – An "Optimal" Key-Comparison Sort

1. Split array into two sub-arrays (unless $n<12$).
2. Call Mergesort recursively for each sub-array.
3. Merge the two ordered sub-arrays.

```
Item *aux;

void mergesortABr(Item a[], Item b[], int ell, int r)
{
  int m = (ell+r)/2;
  if (r-ell <= 10)
  {
    insertion(a, ell, r);
    return;
  }
  mergesortABr(b, a, ell, m);
  mergesortABr(b, a, m+1, r);
  mergeAB(a+ell, b+ell, m-ell+1, b+m+1, r-m);
}

void mergesortAB(Item a[], int ell, int r)
{
  int i;
  for (i = ell; i <= r; i++)
    aux[i] = a[i];
  mergesortABr(a, aux, ell, r);
}
```

How much work (time) in worse case? ($T(n)$ – a *recurrence*)

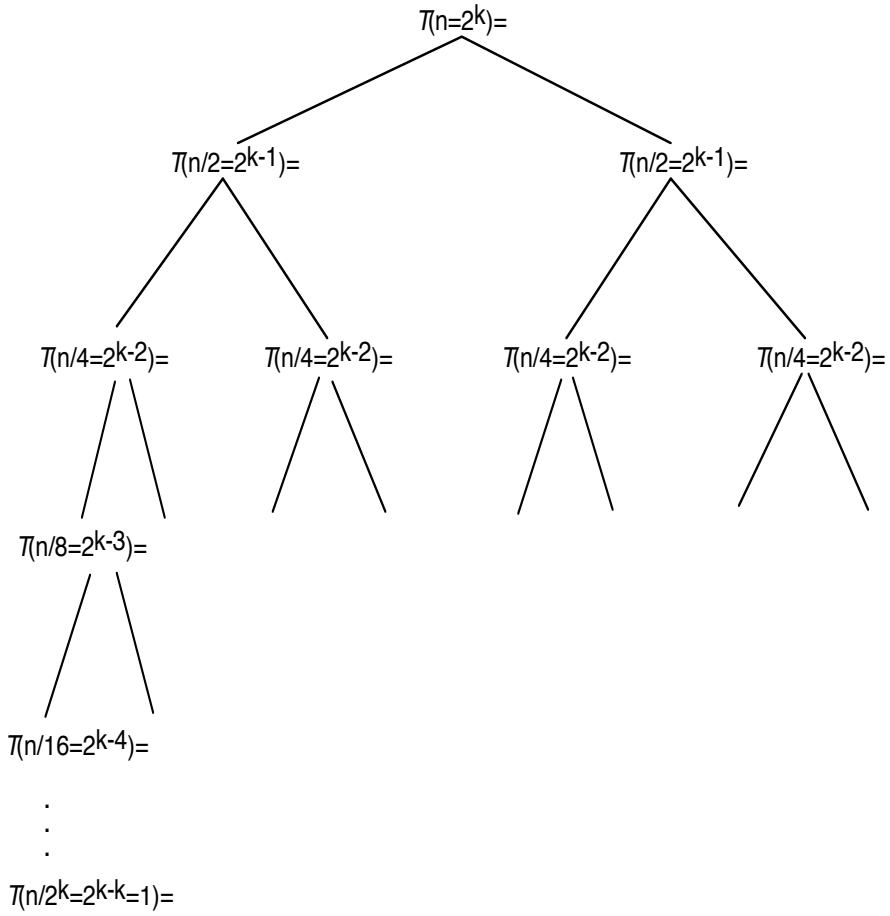
1. Split or call to `insertion()`: constant number of steps.
2. Call recursively:

$$T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + T\left(\left\lceil \frac{n}{2} \right\rceil\right)$$

3. Merge together (n steps)

$$T(n) = c_1 + T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + T\left(\left\lceil \frac{n}{2} \right\rceil\right) + c_2 n = cn \log_? n$$

Recursion Tree



[Don't generalize from this example. More of these later.]

Practical Issues - Sedgewick 8.2, 8.4, 8.5, 8.6

Stability?

These affect the *constants* for *time* and *space*:

1. Merging wastes time checking whether indices have passed the ends of arrays.

(8.2, aside) Store the two sequences as one *bitonic* sequence, then have left and right indices move towards each other.

2. Recursion overhead is wasteful when dealing with small sub-problems.

(8.4) Use a simpler sort when the sub-problem size is below a threshold ("cut-off").

3. The *top-down* recursion is only controlling the number of items in the sub-problems at each level of the recursion tree.

(8.5) Program 8.5 (p. 348) and figure 8.5 (p. 349) demonstrate how to do this *bottom-up* without recursion.

Data Stored as Linked Lists - Sedgewick 8.7

Arrays vs. linked lists tradeoffs.

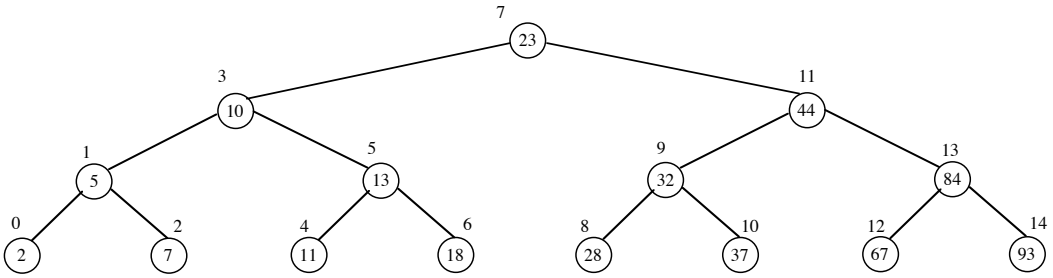
Merge "pattern" applies for linked lists - initialize, neither list exhausted, one list exhausted.

Both top-down and bottom-up mergesorts are very easy to implement.

1.D. BINARY SEARCH - "Optimal" Search of an Ordered Table (or "Space")

Sedgewick 2.6, 12.4

Concept – search *ordered* table in logarithmic time. Consider table with $2^k - 1$ slots.



(http://ranger.uta.edu/~weems/NOTES2320/binarySearch.c)

```
int binSearch(int *a,int n,int key)
// Input: int array a[] with n elements in ascending order.
//        int key to find.
// Output: Returns some subscript of a where key is found.
//         Returns -1 if not found.
// Processing: Binary search.
{
  int low,high,mid;
  low=0;
  high=n-1;
// subscripts between low and high are in search range.
// size of range halves in each iteration.
  while (low<=high)
  {
    mid=(low+high)/2;
    if (a[mid]==key)
      return mid; // key found
    if (a[mid]<key)
      low=mid+1;
    else
      high=mid-1;
  }

  return (-1); // key does not appear
}
```

Recursive binary search? (p. 498)

```
Item search(int ell, int r, Key v)
{ int m = (ell+r)/2;
  if (ell > r) return NULLitem;
  if eq(v, key(st[m])) return st[m];
  if (ell == r) return NULLitem;
  if less(v, key(st[m]))
    return search(ell, m-1, v);
  else return search(m+1, r, v);
}
Item STsearch(Key v)
{ return search(0, N-1, v); }
```

Multiple occurences of keys (http://ranger.uta.edu/~weems/NOTES2320/binarySearchRange.c)

Find i such that a[i-1] < key <= a[i]

```
int binSearchFirst(int *a,int n,int key)
// Input: int array a[] with n elements in ascending order.
//        int key to find.
// Output: Returns subscript of the first a element >= key.
//         Returns n if key>a[n-1].
// Processing: Binary search.
{
  int low,high,mid;
  low=0;
  high=n-1;
// Subscripts between low and high are in search range.
// Size of range halves in each iteration.
// When low>high, low==high+1 and a[high]<key and a[low]>=key.
  while (low<=high)
  {
    mid=(low+high)/2;
    if (a[mid]<key)
      low=mid+1;
    else
      high=mid-1;
  }
  return low;
}
```

Relationship of low and high on return?

Find i such that a[i] <= key < a[i+1]

```
int binSearchLast(int *a,int n,int key)
{
// Input: int array a[] with n elements in ascending order.
//        int key to find.
// Output: Returns subscript of the last a element <= key.
//         Returns -1 if key<a[0].
// Processing: Binary search.
  int low,high,mid;
  low=0;
  high=n-1;
// subscripts between low and high are in search range.
// size of range halves in each iteration.
// When low>high, low==high+1 and a[high]<=key and a[low]>key.
  while (low<=high)
  {
    mid=(low+high)/2;
    if (a[mid]<=key)
      low=mid+1;
    else
      high=mid-1;
  }
  return high;
}
```

Relationship of low and high on return?

Partial output from `binarySearchRange.c` (`count` is `last-first+1`)

```
-- table --         key  first  last  count
  0    0             -1     0    -1     0
  1    1              0     0     0     1
  2    1              1     1     3     3
  3    1              2     4     4     1
  4    2              3     5     4     0
  5    4              4     5     6     2
  6    4              5     7     6     0
  7    6              6     7     9     3
  8    6              7    10     9     0
  9    6              8    10     9     0
 10   10              9    10     9     0
 11   12             10    10    10     1
 12   12             11    11    10     0
 13   12             12    11    14     4
 14   12             13    15    14     0
 15   15             14    15    14     0
 16   15             15    15    16     2
 17   17             16    17    16     0
 18   17             17    17    18     2
 19   18             18    19    19     1
                     19    20    19     0
                     20    20    19     0
```