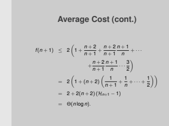
Average Cost (cont.)

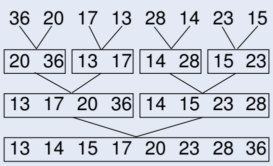
$$\begin{aligned} f(n+1) &\leq 2\left(1 + \frac{n+2}{n+1} + \frac{n+2}{n+1}\frac{n+1}{n} + \cdots \right.\\ &\left. + \frac{n+2}{n+1}\frac{n+1}{n}\cdots\frac{3}{2}\right)\\ &= 2\left(1 + (n+2)\left(\frac{1}{n+1} + \frac{1}{n} + \cdots + \frac{1}{2}\right)\right)\\ &= 2 + 2(n+2)(\mathcal{H}_{n+1} - 1)\\ &= \Theta(n \log n). \end{aligned}$$

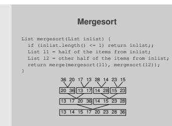


$\mathcal{H}_{n+1} = \Theta(\log n)$

Mergesort

```
List mergesort(List inlist) {
  if (inlist.length() <= 1) return inlist;;
  List l1 = half of the items from inlist;
  List l2 = other half of the items from inlist;
  return merge(mergesort(l1), mergesort(l2));
}
```



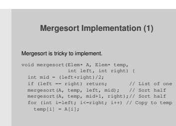


no notes

Mergesort Implementation (1)

Mergesort is tricky to implement.

```
void mergesort(Elem* A, Elem* temp,
               int left, int right) {
  int mid = (left+right)/2;
  if (left == right) return;       // List of one
  mergesort(A, temp, left, mid);   // Sort half
  mergesort(A, temp, mid+1, right);// Sort half
  for (int i=left; i<=right; i++) // Copy to temp
    temp[i] = A[i];
```

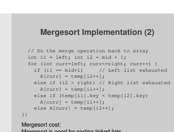



This implementation requires a second array.

Mergesort Implementation (2)

```
  // Do the merge operation back to array
  int i1 = left; int i2 = mid + 1;
  for (int curr=left; curr<=right; curr++) {
    if (i1 == mid+1)      // Left list exhausted
      A[curr] = temp[i2++];
    else if (i2 > right) // Right list exhausted
      A[curr] = temp[i1++];
    else if (temp[i1].key < temp[i2].key)
      A[curr] = temp[i1++];
    else A[curr] = temp[i2++];
}}
```

Mergesort cost:
Mergesort is good for sorting linked lists.



Mergesort cost: $\Theta(n \log n)$

Linked lists: Send records to alternating linked lists, mergesort each, then merge.

Heaps

Heap: Complete binary tree with the **Heap Property**:

- Min-heap: all values less than child values.
- Max-heap: all values greater than child values.

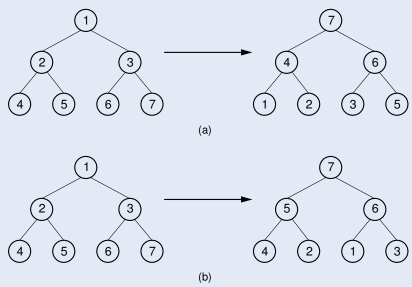
The values in a heap are **partially ordered**.

Heap representation: normally the array based complete binary tree representation.

2014-02-12
CS 5114
└ Heaps

no notes

Building the Heap



(a) requires exchanges (4-2), (4-1), (2-1), (5-2), (5-4), (6-3), (6-5), (7-5), (7-6).
(b) requires exchanges (5-2), (7-3), (7-1), (6-1).

2014-02-12
CS 5114
└ Building the Heap

This is a Max Heap
How to get a good number of exchanges? By induction.
Heapify the root's subtrees, then push the root to the correct level.

Siftdown

```
void heap::siftdown(int pos) { // Sift ELEM down
  assert((pos >= 0) && (pos < n));
  while (!isLeaf(pos)) {
    int j = leftchild(pos);
    if ((j<(n-1)) &&
        (Heap[j].key < Heap[j+1].key))
      j++; // j now index of child with > value
    if (Heap[pos].key >= Heap[j].key) return;
    swap(Heap, pos, j);
    pos = j;  // Move down
  }
}
```


2014-02-12
CS 5114
└ Siftdown

no notes

BuildHeap

For fast heap construction:

- Work from high end of array to low end.
- Call `siftdown` for each item.
- Don't need to call `siftdown` on leaf nodes.

```
void heap::buildheap()        // Heapify contents
  { for (int i=n/2-1; i>=0; i--) siftdown(i); }
```

Cost for heap construction:

$$\sum_{i=1}^{\log n} (i-1)\frac{n}{2^i} \approx n.$$

2014-02-12
CS 5114
└ BuildHeap

$(i-1)$ is number of steps down, $n/2^i$ is number of nodes at that level.

The intuition for why this cost is $\Theta(n)$ is important. Fundamentally, the issue is that nearly all nodes in a tree are close to the bottom, and we are (worst case) pushing all nodes down to the bottom. So most nodes have nowhere to go, leading to low cost.

Heapsort

Heapsort uses a max-heap.

```
void heapsort(Elem* A, int n) { // Heapsort
  heap H(A, n, n);              // Build the heap
  for (int i=0; i<n; i++)       // Now sort
    H.removemax(); // Value placed at end of heap
}
```

Cost of Heapsort:

Cost of finding k largest elements:

Cost of Heapsort: $\Theta(n \log n)$
Cost of finding k largest elements: $\Theta(k \log n + n)$.

- Time to build heap: $\Theta(n)$.
- Time to remove least element: $\Theta(\log n)$.

Compare Heapsort to sorting with BST:

- BST is expensive in space (overhead), potential bad balance, BST does not take advantage of having all records available in advance.
- Heap is space efficient, balanced, and building initial heap is efficient.

Binsort

A simple, efficient sort:

```
for (i=0; i<n; i++)
  B[key(A[i])] = A[i];
```

Ways to generalize:

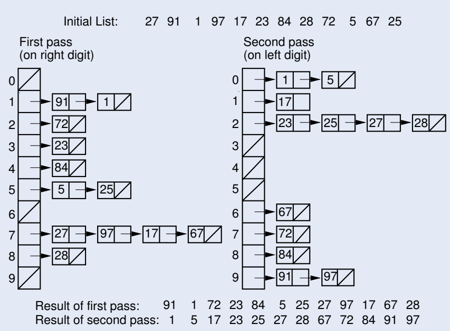
- Make each bin the head of a list.
- Allow more keys than records.

```
void binsort(ELEM *A, int n) {
  list B[MaxKeyValue];
  for (i=0; i<n; i++) B[key(A[i])].append(A[i]);
  for (i=0; i<MaxKeyValue; i++)
    for (each element in order in B[i])
      output(B[i].currValue());
}
```

Cost:

The simple version only works for a permutation of 0 to $n-1$, but it is truly $O(n)$!
Support duplicatesI.e., larger key spaceCost might look like $\Theta(n)$.
Oops! It is ctually, $\Theta(n * \text{Maxkeyvalue})$.
Maxkeyvalue could be $\mathrm{O}(n^2)$ or worse.

Radix Sort



no notes

Radix Sort Algorithm (1)

```
void radix(Elem* A, Elem* B, int n, int k, int r,
           int* count) {
  // Count[i] stores number of records in bin[i]

  for (int i=0, rtok=1; i<k; i++, rtok*=r) {
    for (int j=0; j<r; j++) count[j] = 0; // Init

    // Count # of records for each bin this pass
    for (j=0; j<n; j++)
      count[(key(A[j])/rtok)%r]++;

    //Index B: count[j] is index of j's last slot
    for (j=1; j<r; j++)
      count[j] = count[j-1]+count[j];
```

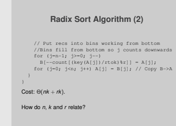

no notes

Radix Sort Algorithm (2)

```
    // Put recs into bins working from bottom
    //Bins fill from bottom so j counts downwards
    for (j=n-1; j>=0; j--)
      B[--count[(key(A[j])/rtok)%r]] = A[j];
    for (j=0; j<n; j++) A[j] = B[j]; // Copy B->A
  }
}
```

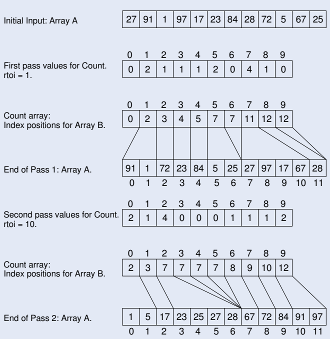
Cost: $\Theta(nk + rk)$.

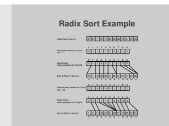
How do n, k and r relate?



r can be viewed as a constant.
$k \geq \log n$ if there are n distinct keys.

Radix Sort Example





no notes

Sorting Lower Bound

Want to prove a lower bound for *all possible* sorting algorithms.

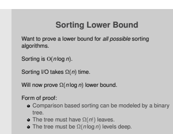
Sorting is $O(n \log n)$.

Sorting I/O takes $\Omega(n)$ time.

Will now prove $\Omega(n \log n)$ lower bound.

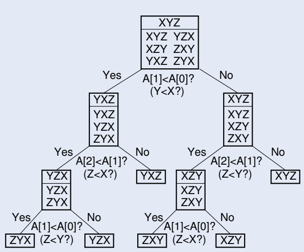
Form of proof:

- Comparison based sorting can be modeled by a binary tree.
- The tree must have $\Omega(n!)$ leaves.
- The tree must be $\Omega(n \log n)$ levels deep.

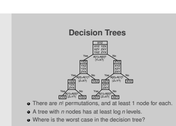


no notes

Decision Trees



- There are $n!$ permutations, and at least 1 node for each.
- A tree with n nodes has at least $\log n$ levels.
- Where is the worst case in the decision tree?



no notes

Lower Bound Analysis

$$\log n! \leq \log n^n = n \log n.$$

$$\log n! \geq \log \left(\frac{n}{2}\right)^{\frac{n}{2}} \geq \frac{1}{2}(n \log n - n).$$

- So, $\log n! = \Theta(n \log n)$.
- Using the decision tree model, what is the average depth of a node?
- This is also $\Theta(\log n!)$.

$\log n -$ (1 or 2).