A Summary of Sorting

by Marius Aamodt Eriksen (marius@umich.edu)

General Characteristics

A couple of properties apply to all sorts;

1. A sort is called *stable* if it preseres the original content and order of the data, only producing a different set of keys.
2. Indirect sorts are done in terms of referal; either through index values or pointers.
3. Adaptive sorts take into account the nature of the data, and is able to adapt on-the-fly decreasing the general complexity of the algorithm.

Insertion Sort

Insertion sort builds on the concept of "where to insert the next item." That is to say, going through an ever expaning range (within a subrange), place each item in order. An optimized insertion sort algorithm follows:

```
/* place the smallest element to the very left (lowest part) of the range */
for (int i = right; i > left; i--) if (a[i-1] > a[i]) switch(a[i-1], a[1]);
for (i = left+2; i <= right; i++) {
   int j = i;
   tp t = a[j];
   /* shift (and swap) down until we've reached the right place */
   while (t < a[j-1]) {
      a[j] = a[j-1];
      j--;
   }
   /* insert our value */
   a[j] = t;
}
```

This version of the insertion sort is adapteable, i.e. it will move to the next iteration once it has found the right spot to insert. Here, `left` and `right` are the boundaries of the sort on the matrix `a`. The insertion sort has a complexity of $O(N^2)$ to sort N items. This, however, also depends greatly on the initial order of the data.

Selection Sort

Selection sort operates on a very simple principle, first finding the smallest element in the set, exchaning that with the first position, then going on to the second smallest, exchanging that with the second position in the set, and so on and so forth. It is so named, because it repeatedly *selects* the smallest item in the remaining set. A generic algorithm follows:

```
for (int i = left; i < right; i++) {
   int min = i;
      for (int j = i+1; j <= right; j++
         if (a[j] < a[min]) min = j;
      switch(a[i], a[min]);
}
```

The selection sort depends little on the pre-existing order of the data set, which may make it an undesirable algorithm to choose.

Bubble Sort

Bubble sort, while being extremely easy to implement, is not the most efficient of sorts. It works on the principle of picking a value, then exchanging it "down" until it is in order (that is to say, the previous element is smaller). Continuing this, with decreased ranges will eventually produce a sorted set. A sample bubble sort is:

```
for (int i = left; i < right; i++)
   for (int j = right; j > i; j--)
      if (a[j-1] > a[j]) switch(a[j-1], a[j]);
```

A quite obvious, but also inefficient algorithm.

Summary of Bubble, Selection and Insertion Sorts

All of the above mentioned algorithms run in $O(N^2)$ (quadratic) differing only by a constant. This is true both in the average and worst-case scenarios.

Shellsort

Shellsort is essentially an extension of insertion sort that is more efficient. By considering every h items, instead of every adjacent item, and have several segments, one is able to move objects larger distances (not just adjacent objects). This makes insertion sort more efficient when objects are largely out of order. The generic algorithm is largely similar to the insertion sort, except that h needs to be determined, and h elements are decremented at every iteration of the inner loop, instead of 1.

```
for (int h = 1; h <= (r-1)/9; h=3*h+1);
for (; h > 0; h /= 3)
   for (int i = l+h; i <= r; i++) {
      int j = i;
      te t = a[j];
      while (j >= l+h && t < a[j-h]) {
         a[j] = a[j-h];
         j -= h;
      }
      a[j] = t;
}
```

This particular implementation (Sedgewick) uses the ratio of roughly $\frac{1}{3}$.

Quicksort

Certainly one of the more well-studied algorithms, quicksort performs an average case of $O(N \log N)$, requiring a worst case of $O(N^2)$. Quicksort works on the traditional divide-and-conquer strategy, partitioning the data into independent pieces that need to be sorted. The partitions must meet the requirements:

1. The element `a[i]` is in its final position in the array.
2. All of the elements, `a[l], ..., a[i-1]` are smaller than `a[i]`.
3. All of the elements, `a[i+1], ..., a[r]` are greater than `a[i]`.

where `l` stand for the left (lower extreme), and `r` for the right (upper extreme). The generic algorithm looks like this:

```
void qsort(tp a[], int l, int r) {
   if (r <= l) return;
   int i = partition(a, l, r);
   qsort(a, l, i-1);
   qsort(a, i+1, r);
}
```

It is clear that the current range needs to be partitioned, accordnig to the rules above, then quicksort exposed again on the two resulting ranges. The algorithm is recursive in nature.

One strategy to implement sorting involves taking `a[r]` as a "partitioning element." Going from left to right, continuing until an element that is greater than our partitioning elemtent is found. Then going from right to left, finding an element that is smaller than the partitioning element. Both of these are out of place, and can thus be exchanged. Iterating over this, assures us that anything below the lower pointer is smaller than the partitioning element, and above the upper pointer, larger. A sample implementation of `partition()` follows:

```
int partition(tp a[], int l, int r) {
   int i = l-1, j = r;
   tp t = a[r];
   for (;;) {
      while (a[++i] < t);
      while (t < a[--j]) if (j == l) break;
      if (i >= j) break;
      switch(a[i], a[j]);
   }
   switch(a[i], a[r]);
   return i;
}
```

Clearly, a worst-case quicksort would be when all the elements are already sorted, then the algorithm would require $O(\frac{N^2}{2})$ time. The best-case scenario of a quicksort is when the partitioning scheme used divides the set exactly in half. This happens *on the average*, it's therefore safe to say that quicksort performs $O(N \lg N)$ on the average.

Other improvement techniques also exists. For example, since the quicksort is guaranteed to address a number of smaller subfiles (in fact, a large number of them), we can find a better method of sorting those, for example using insertion sort. In such a case, a threshold value for size would have to be set. One could also choose to *not* sort subfiles that have sizes smaller than this threshold, leaving the result an almost-sorted file. An insertion sort, ideal for almost-sorted files, could then be called on the remaining file.

Other strategies involve using the median of three of the variables in the file. On the average, this should produce a good split.

Mergesort

Mergesort is a sorting algorithm that is capable of merging two ordered files into one ordered file. The algorithm is quite simple, inserting the smaller of the two elements into the resulting set until they are both exhausted. If one gets exhausted sooner than the other, elements are just taken from the other set. A sample algorithm is:

```
for (int i = 0, int j = 0, int x = 0; x < N+M; x++ ) {
   if (i==N) { c[k]=b[j++]; continue; }
   if (j==M) { c[k]=a[i++]; continue; }
   if (a[i] < b[j])
      c[k] = a[i++];
   else
      c[k] = a[j++];
}
```

Here, N and M are the sizes of the a and b arrays respectively.