1 Algorithms and runtime

1.1 Min,Max

Finding the maximum entry in array A, returning maximum value and location.

```
findMax(iMax,theMax,A,i,j) {
  iMax = i; theMax = A[i];
  for (k=i+1..j) if ( theMax < A[k] ) { iMax = k; theMax = A[k]; }
}
```

runtime is $O(j-i)$ (linear)

Finding the minimum entry in array A, returning minimum value and location.

```
findMin(iMin,theMin,A,i,j) {
  iMin = i; theMin = A[i];
  for (k=i+1..j) if ( theMin > A[k] ) { iMin = k; theMin = A[k]; }
}
```

runtime is $O(j-i)$ (linear)

1.2 Sorting

Sort an array A of numerical values in non-decreasing order.

```
QuadSortIncrease(A,i,j) {
  if ( i < j ) {
     findMax(iMax,theMax,A,i,j);
     A[iMax] = A[j]; A[j] = theMax;
     QuadSortIncrease(A,i,j-1);
  }
}
```

runtime is $O((j-i)^2)$. (quadratic)

Merge two sorted arrays B and C into new sorted array A.

```
Merge(A,ia,ja,B,ib,jb,C,ic,jc) {
  B[jb+1] = infty; C[jc+1] = infty;
  bf = ib; cf = ic;
  for (af=ai..aj) {
```

```
    if ( B[bf] < C[cf] ) { A[af] = B[bf]; bf = bf+1; }
    else                 { A[af] = C[cf]; cf = cf+1; }
  }
}
```

runtime is $O(ja - ia)$ (linear)

```
QuickerSortInc(A,i,j) {
  if ( i < j ) {
     k = (i+j)/2 ;
     QuickerSortInc(A,i,k);
     QuickerSortInc(A,k+1,j);
     Merge(A,i,j, A,i,k, A,k+1,j);
  }
}
```

runtime is $O((j-i)log(j-i))$ (loglinear)

1.3 SubsetSum

List all possible subsetsums from elements in array A:

```
FullEnumeration(B,A,n) {
  for (k=1..2^n) { B[k] = 0; }
  for (m=1..n) {
    for (k=1..2^(m-1)) { B[k+2^(m-1)] = B[k]+A[m]; }
  }
}
```

runtime is $O(2^n)$ (exponential)

Generate the listing in non-decreasing order:

```
FullEnumSorted(B,A,n) {
  for (k=1..2^n) { B[k] = 0; }
  for (m=1..n) {
    for (k=1..2^(m-1)) { B[k+2^(m-1)] = B[k]+A[m]; }
    Merge(B,1,2^m, B,1,2^(m-1), B,2^(m-1)+1,2^m);
  }
}
```

runtime remains $O(2^n)$ (exponential)

Detect whether there is a subset of elements in array A adding up to target value L.

```
findSubsetSum(det,A,n,L) {
  FullEnumeration(B,A,n) ;
  det = FALSE;
  for (k=1..2^n) if ( B[k] == L ) { det = TRUE; }
}
```

runtime is $O(2^n)$ (exponential)

Find subset of elements of array A adding up to L:

```
findSubset(det,S,A,n,L) {
  FullEnumeration(B,A,n) ;
  det = FALSE; S = {};
  for (k=1..2^n) if ( B[k] == L ) { det = TRUE; pos = k; }
  if (det == TRUE ) {
     while ( pos > 1 )  {
        k = ceil( log(pos) );  // base-2 logarithm rounded up
        S = S + {k};
        pos = pos - 2^(k-1);
     }
  }
}
```

runtime is $O(2^n)$ (exponential)

```
HalfEnum(det,A,n,L) {
  Copy(B,1,n/2, A, 1,n/2);
  Copy(C,1,n-(n/2), A,(n/2)+1,n);
  FullEnumSorted(BS,B,n/2) ;
  FullEnumSorted(CS,C,n-(n/2)) ;
  det = FALSE; bf = 1; cf = 2^(n-(n/2));
  done = FALSE;
  while not done {
    if ( BS[bf]+CS[cf] == L ) {
       det = TRUE; done = TRUE; Lb = BS[bf]; Lc = CS[cf]; }
    }
    else {
       if ( BS[bf]+CS[cf] < L ) { bf = bf+1; }
       else                     { cf = cf-1; }
    }
    if ( cf == 0 OR bf > 2^(n/2) ) { done = TRUE; }
  }
}
```

runtime is $O(2^{(n/2)})$ (half-exponential). Note that if subset is detected it can be traced back by `findSubset(det,Sb,B,n/2,Lb)` and `findSubset(det,Sc,C,n-(n/2),Lc)`.

2 Recurrence and Dynamic Programming

Some functions in the first section recursively call themselves, on smaller inputs. This may lead to easily formulated definitions. In the above examples the recursion is on subsets. We can also do this on the NUMBERS.

2.1 Recursion

Let A be an array of n positive integer numbers, with $B = \sum_i A[i]$. To answer the subsetSum question for integer $L < B$, we may also use the following procedure:

```
TraceRecursive(det, A,n,L) {
  if ( n == 1 ) {
     if ( A[1] == L ) then  { det = TRUE; }
     else                   { det = FALSE; }
  }
  else {
     TraceRecursive(det1, A,n-1,L);
     TraceRecursive(det2, A,n-1,L-A[n]);
     det = ( det1 OR det2 );
  }
}
```

The runtime will be $O(2^n)$. The procedure is slightly better when we detect negative targets:

```
TraceRecursivePos(det, A,n,L) {
  if ( n == 1 ) {
     if ( A[1] == L ) then  { det = TRUE; }
     else                   { det = FALSE; }
  }
  else {
     TraceRecursivePos(det1, A,n-1,L);
     det2 = FALSE;
     if ( L > A[n] ) { TraceRecursive(det2, A,n-1,L-A[n]); }
     det = ( det1 OR det2 );
  }
}
```

Note that we actually may compute multiple times the expression `TraceRecursivePos(det, A,10,L-1000)` if it is the case that different combinations of elements from $A_{11}, A_{12}, ..., A_n$ happen to add up to 1000.

2.2 Dynamic Programming

Making use of the fact that there are only a few possible values for the last argument, namely $0, \ldots, L$, we run the same procedure but compute `TraceRecursivePos(det, A,i,M)` for

small values of i and M FIRST. Let $B(i, M) = 1$ if `TraceRecursivePos(det, A,i,M)` yields `det=TRUE` and $B(i, M) = 0$ if `TraceRecursivePos(det, A,i,M)` yields `det=FALSE`. Here we even allow for $i = 0$.

```
DynProgTrace(det,B,A,n,L) {
  B[0,0] = 1;
  for (j=1..L) { B[0,j] = 0; }
  for (i=1..n) {
     for (j=0..L) {
        B[i,j] = 0;
        if ( j-A[i]>=0 AND B[i-1,j-A[i]]==1 ) { B[i,j] = 1; }
        if ( B[i-1,j] == 1 )                  { B[i,j] = 1; }
     }
  }
  det = FALSE;
  if ( B[n,L] == 1 ) { det = TRUE; }
}
```

Note that the running time is now $O(nL)$, since it is a constant amount of work for filling one entry in table B.

Also note that a partial solution with $B[i, j] = 1$ can be extended to $B[i + 1, j + A[i + 1]] = 1$ WITHOUT explicitly knowing how solution $B[i, j]$ is built up. The system is MEMORYLESS.

2.3 Knapsack

In the knapsack problem we are given n items i with a certain (integer) weight $A(i)$, and a positive value $V(i)$. We want to pack a subset of highest total value in our knapsack of limited capacity K (integer). The above mentioned procedure is easily modified into an optimization routine of runtime $O(nK)$.

```
DynProgKnapsack(maxVal,B,A,V,n,K) {
  for (j=0..K) { B[0,j] = 0.0; }
  for (i=1..n) {
     for (j=0..K) {
        B[i,j] = B[i-1,j];
        if ( j-A[i]>=0 AND B[i-1,j-A[i]] + V[i] > B[i,j] ) {
           B[i,j] = B[i-1,j-A[i]] + V[i];
        }
     }
  }
  maxVal = B[n,0];
  for (j=1..K) if ( B[n,j] > maxVal ) { maxVal = B[n,j]; }
}
```

runtime is again $O(nK)$, and the optimal selection of items can be traced back using matrix B.

3 Local Search

In the previous cases we find an optimal solution by explicit or implicit total enumeration. If the total solution space is too big we may accept a sub-optimal solution. One way to find such solution is pick an arbitrary solution, and from that start point go on a journey through solution space, by hopping from one solution to a *neighboring* one.

3.1 Neighborhood of a solution

In a quite general setting we consider a solution to a combinatorial problem. For the particular problem at hand we may be able to define a *move* which turns the current solution into another one. For instance, for the 100 number problem where we want a partition into ten stacks of almost equal sum, we may consider

moveOne move one item from its current group to another group

exchange2 select two items from different stacks and interchange them

exchange2groups select two groups of items from different stacks and interchange the groups

Now we can define the *neighborhood* $N(S)$ of solution S as the set of all feasible solutions that can be the result of a move applied to S.

3.2 Neighbor selection

In some cases we can systematically scan the whole neighborhood $N(S)$ of S for a better solution. If we find such (strictly) better solution S' we may replace S by S', and repeat from there. We may continue until no element in the current neighborhood yields another improvement. This method is known as *iterative improvement*. It ends at a *local optimum*.

Another way can be to randomly select a neighbor and test whether it is an improvement. Again only accept proper improvements. This method should be terminated by a bound on the number of trials or on running time.

3.3 Repeated search

The above iterative improvement scheme may be started from a randomly generated starting solution. If this is the case we may repeat the process after a local optimum has been reached. Of course we keep track of the overall best solution found so far.

Another method of getting out of the local optimum can be to accept - under some conditions - a move towards a worse solution. Mechanisms that employ this are *Simulated Annealing*, *Tabu Search* and *Variable Depth Search*. Finally this walk through solution space can also be applied to a *population of solutions*. In this case we need mechanisms to move from one population to another. This approach is called *Genetic Algorithm* or *Evolutionary Algorithm*, because it is mimicing evolution in terms of mutation and mating.