Unit 2 Notes

New Bentley Optimization Rules

Writing Efficient Programs by Jon Louis Bentley

In the new set of Bentley rules, the optimization is applied only on work no architecture-dependent optimizations.

Data Structures

Packing and Encoding

Packing: Store more than one data value in a machine word.
Encoding: Convert data values into a representation requiring fewer bits.

Encoding a date into a single word can make representing a date in less bits but it requires more computation if certain operation need to run on these encoded dates.

Packing a date into o struct still only takes less than a word, but with more quickly extraction for individual fields.

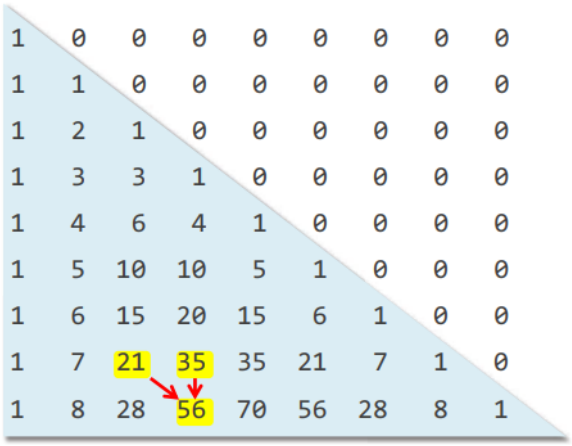
Φ *Sometimes unpacking and decoding are the optimization, depending on whether more work is involved moving the data or operating on it.*

Augmentation

Add information to a data structure to make common operations do less work. For example, if we only save the head for a list, appending a list will require traversal of the first list to the end in order to get the address of the last element. In this case, we can augment the list with a tail pointer which helps operate list appending in constant time.

Precomputation

To perform calculations in advance and avoid doing them at mission-critical times. Similar to Dynamic Programming. For example, Pascal's Triangle table for Binomial Coefficients $\binom{n}{k} = \frac{n!}{k!(n-k)!}$.



We still need to compute the table but we can put the table in source code which can be done in compile time.

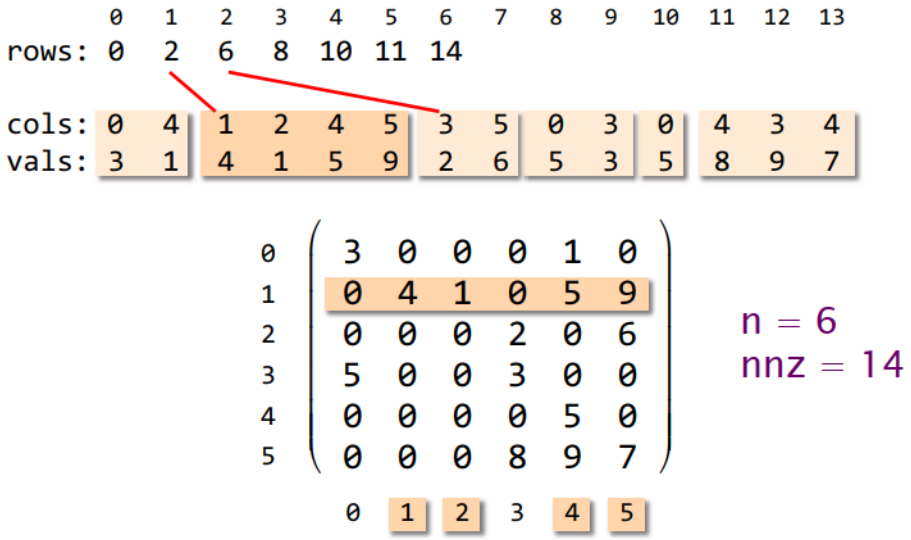
Caching

Store result that have been access recently so that the program need not compute them again.

Sparsity

Avoid storing and computing on zeros since any operation with 0 can be determined before computation.

Use Compressed Sparse Row (CSR)



Storage is O(n+nnz) instead of n^2

Logic

Constant Folding and Propagation

Evaluate constant expressions and substitute the result into further expressions, all during compilation.

Common-Subexpression Elimination

To avoid computing the same expression multiple times by evaluating the expression once and storing the result for later use.

Algebraic Identities

Replace expensive algebraic expressions with algebraic equivalents that require less work. For example, square root is expensive so that we can use power to replace it just by taking the power of two for both side of the calculation.

$\sqrt{u} \leq v$ exactly when
$u \leq v^2$.

Short Circuiting

When performing a series of tests, the idea of short-circuiting is to stop evaluating as soon as you know the answer.

Ordering Test

A sequence of logical tests. If we can expect that most of the statement will be pass, we should put the one most possible to fail earlier and vice versa. Additionally, if a certain test will take a lot of computation, it should be put back as well.

Combining Tests

Combining a sequence of test with one test or switch.

Loops

Hoisting

Avoid recomputing loop-invariant code each time through the body of a loop.

```c
#include <math.h>

void scale(double *X, double *Y, int N) {
  for (int i = 0; i < N; i++) {
    Y[i] = X[i] * exp(sqrt(M_PI/2));
  }
}
```

```c
#include <math.h>

void scale(double *X, double *Y, int N) {
  double factor = exp(sqrt(M_PI/2));
  for (int i = 0; i < N; i++) {
    Y[i] = X[i] * factor;
  }
}
```

Sentinels

Sentinels are special dummy values placed in a data structure to simplify the logic of boundary conditions, and in particular, the handling of loop-exit tests.

```c
bool overflow(int64_t *A, size_t n) {
    //All elements of A are nonegative
    int64_t sum = 0;
    for(size_t i = 0; i < n; ++i) {
        sum += A[i];
        if(sum < A[i]) return true;
    }
    return false;
}
```

In the above code, each loop will have to do two tests. One on the size of I and the other on sum overflow. We can modify this code so that we are able to do only one check in each loop.

```c
bool overflow(int64_t *A, size_t n) {
    //All elements of A are nonegative
    A[n] = INT64_MAX;
    A[n + 1] = 1;
    size_t i = 0;
    int64_t sum = A[0];
    while (sum >= A[i]) {
        sum += A[++i];
    }
    if(i < n) return true;
    return false;
}
```

Loop Unrolling

Save work by combining several consecutive iterations of a loop into a single iteration. Reducing the total number of iterations of the loop and the number of times that the instructions that control the loop must be executed. Additionally, this enables more compiler optimizations since compiler now have more code to work with. However, if the loop body is already huge, loop unrolling will not have much effect on performance.